**Designing a Real Time Application using UML**

**Abstract:** The paper explains the importance of constructing an architectural model to help in designing a real-time embedded system. The embedded real-time software encountered in applications such as automobiles, telecommunications, aerospace and domestic appliances tend to be large and extremely complex. A good architectural design accommodates changes forced by the set of new requirements. In order to tailor UML towards real-time domain, extendibility mechanisms such as profiles and stereotypes are used. Design patterns are introduced to model the safety critical architecture of the system being built. An example of a real-time smoke detector system is shown using UML.

**1. Introduction to Real Time Systems Application Domain**

The one common feature of all real-time software systems is timeliness, that is the requirement of the system to respond correctly to inputs within acceptable time intervals. However, this simple property characterizes a vast spectrum of very different types of systems ranging from purely time-driven to purely event-driven systems, from soft real-time systems to hard real-time systems, and so on. Each of these categories of systems has developed its own syntax, design patterns, and modeling styles that collectively capture the distinctive experience of various real-time projects.

This document focuses on a major category of real-time systems that are characterized as complex, distributed, event or time driven. Such systems are frequently encountered in telecommunications, aerospace, defense, and automatic control applications. The size and complexity of these systems demand a considerable initial development effort, typically involving large development teams, that is followed by an extended period of evolutionary growth. During this time new requirements are identified and the system is modified incrementally to meet them.

Under these circumstances the primary concern for the developers is the architecture of the software which refers to the essential structural and behavioral framework on which all other aspects of the system depend. The above mentioned architecture is the software equivalent of building a large infrastructure where any changes to the foundation necessitates complex and costly changes to substantial parts of the system. Therefore, a well-designed architecture is not only one that simplifies construction of the initial system, but more importantly, one that easily accommodates changes forced by new system requirements.
2. Use of UML in Real Time Area

Some of the models provided by the UML for specifying general systems are

1. Use cases and scenarios
2. Object and class models
3. State charts and other behavioral specifications
4. Design patterns
5. Extensibility mechanisms

Use cases show the communication of the system with external actors in the performance of a system function. Use cases may associate with external actors and may use or extend another use case. When actors communicate with a use case, it means that the actors send or receive messages from the system.

Scenarios show instances of use cases applied in various circumstances. A scenario is a specific, step-by-step sequence of messages sent between the actors and the system. The UML provides two notations for showing scenarios: the sequence diagram and the collaboration diagram. The sequence diagram represents objects (shown as labeled vertical lines) sending messages (shown as horizontal or diagonal arrows).

Class diagrams contain primarily classes, objects, and their relationships. Classes are depicted in these diagrams as rectangles containing the name of the class. This rectangle may be compartmentalized with the class name in the first compartment, attributes in the second compartment, operations in the third, exceptions in the fourth, and so on. Object names are followed by a colon and are underlined.

An important subset of all classes has behavior that can be modeled using finite state machines. Such classes are called reactive because they react in specific ways to incoming events. Reactive classes must always be in exactly one state. When they're constructed, they must enter a valid initial state, though this state may change based on its constructor's parameters.

A design pattern is a kind of template for a solution to a common type of problem. The UML shows design patterns using class diagrams with the addition of an oval denoting the pattern. Dependencies from the oval to the classes identify the roles the classes play within the design pattern. Some of the other specialized extensions provided to the UML are adding action semantics feasibility, executable UML, time analysis, quality of service analysis, schedulable and performance analysis.

Stereotypes are specialization extensions of the UML metamodel itself. A stereotype is the type of the modeling element within the UML metamodel. This means that the user can take a metamodel element, such as the modeling elements class and package, and subclass them.
3. Real Time Problem

The real-time system that is being modeled here is a smoke detector. Smoke detector is one of those interesting inventions that practically cost a little. All smoke detectors consist of two basic parts, a sensor to sense the smoke and a very loud electronic horn to alert people. There are two kinds of smoke detectors namely photo electric and ionization detectors. The normal detector has laser light source on one side and photo beam detector on the other side as shown in the figure 1-A below. When the smoke in the room gets so dense that it blocks the light beam sufficiently, the alarm bell attached to the detector would go off. In the photo electric detector the light source and the beam detector are fixed in perpendicular way to each other so that when the smoke in the room gets denser it scatters the light towards to the beam detector and the alarm would go off as shown in figure 1-B below.

Besides the normal equipment provided for the operation of the smoke detector system, we can attach additional equipment like a communication channel that sends a signal to the fire department that smoke is present in the particular house and a watch dog that can keep a watch on the alarm and the communication channel to check whether they are satisfying the functionality they normally provide.

simple smoke detector system

Figure 1 – A

Figure 1-B

A use case can be used to describe a systems functional requirement and treats the system as a black box. Use cases communicate with some set of the identified actors, and are shown as named ovals on the use case diagram. Actors are represented using stick figures. Use cases may also extend (specialize) other use cases or use the facilities provided by another. Figure 2 below depicts the use case diagram for the smoke detector system.
**Figure 2 use case diagram**

- **Smoke**
  - Home owner
  - Fire personnel

**Figure 3 class diagram**

**Watchdog class**
- check: int // is memory mapped to an area which takes the ANDed value from the alarm and the communicate channel
- checkchannel(): void // compares the value of check and if zero calls the system resetblink() function

**Reset class**
- takevalue: bool // is memory mapped which takes the return value of pressed()
- count: int // time to wait
- disablealarm(): void // calls the system sleepalarm() based on the takevalue

**Alarm class**
- readvalue: int // is memory mapped to an area which takes the value from lightsource()
- avgvalue: int
- soundalarm(): void // calls the system ringalarm() and then either calls the reset or communicate class

**Communicate class**
- address: String // address to send
- send(): void // calls the system communication to send the home owners address to the fire department

**Operating system class**
- lightsource(): void // writes the value of light intensity to a memory location
- resetblink(): void // resets the entire system
- channelalert(): void // writes value 1 or 0 based on whether the alarm and communicate are working to memory
- ringalarm(): void // horns the alarm to the home owner
- sleepalarm(): void // disables the alarm for a specific period
- pressed(): void // writes 0 or 1 to a memory location based on if the reset is pressed or not
- communication(address): void // sends the home owners address to the fire department through the communication channel
The natural place to start object analysis is with the class diagram. The major activities are the identification of the objects and classes, clarification of the responsibilities of the identified objects (and the primary attributes and operations of the objects that will be used to achieve those responsibilities), and identification of the relationships and the properties of those relationships. Basically the smoke detector system has the following classes: 1)alarm class 2)reset class 3)communicate class 4)watchdog class and 5)operating system class and is depicted in **figure 3** above.

In the above class diagram each alarm class is associated with a reset class and a communicate class. The alarm class has a function called soundalarm( ) that sounds the alarm based on the average value of the reading, stokes the watch-dog and informs the fire department as well as home owner. Based on the reading it would either call the communicate class directly or will call the reset class.

The reset class has an operation disablealarm( ) that makes the alarm to stop for some time after the user presses the reset button. If the reset is not pressed after a certain time then it would call the communicate class to inform the fire department. The communicate class has an operation send() that would send the address information of the home owner to the fire department as well as stokes the watch-dog. This class operation is called when ever the average reading in the alarm class is greater than some threshold value or when the user does not press the reset button after some time in the reset class.

The watchdog class calls the system based channelalert( ) function to check whether both the communication channel and the alarm are functioning properly. It does that periodically, once in every sixty seconds. It first pings the communicate channel to see whether the communicate is able to get the IP address of the fire department and then invokes the alarm to see whether it can ring in normal mode. Based on the Anded values, if the check variable turns out to be zero then it calls the system function resetblink( ) that blinks a light for the user saying that some thing is wrong and then resets the entire system.

The operating system class contains all the system built functions that invoke the various hardwares and returns the appropriate values. It has the following functions which do the respective operations: 1) lightsource() it returns the current reading of light density in integer value. 2) ringalarm() rings the alarm for the home owner based on the average value. 3) sleepalarm() disables the alarm for a some time based on the reset value. 4) communication(address) sends the home owners address to the fire department, the other system functions are explained in the preceding paragraph.

Scenarios show instances of use cases applied in various circumstances. It shows a particular interactive sequence of message exchanges between the use case and the participant actors. Each use case represents an essentially infinite set of different scenarios. Fortunately, only a few importantly different scenarios must be explicitly modeled because the others will be trivial variants of this small set. The **figure 4 –A** below depicts a scenario where the alarm has been set off and is informed to the home owner and **figure 4-B** shows fire department being notified by the communicate class.
In this scenario diagram the main object of the system program containing the alarm, reset and communicate class keeps on reading the value of the system light source function. When it gives some value it takes the average of that for hundred readings and then checks for two cases one where it is greater than five and less than ten and one where it is greater than ten. In the first case it calls the reset class to disable the alarm if the user has pressed the reset button. So the reset class calls the OS class for the pressed value, if the user has not pressed the button with in some time then it calls the communicate class, in case where he has pressed then it calls the system sleep alarm function. The home owner is alarmed during this process.

In the second case where the value is greater than ten then it calls the communicate class directly. The communicate class sends the address to the fire department. The watch dog class checks whether the alarm and communicate channel are functioning properly.
In this scenario diagram the main object of the system program containing the alarm, reset and communicate class keeps on reading the value of the system light source function. When it gives some value it takes the average of that for hundred readings and then checks for two cases one where it is greater than five and less than ten and one where it is greater than ten. In the first case it calls the reset class to disable the alarm if the user has pressed the reset button. So the reset class calls the OS class for the pressed value, if the user has not pressed the button with in some time then it calls the communicate class, in case where he has pressed then it calls the system sleep alarm function. In the second case where the value is greater than ten then it calls the communicate class directly. The communicate class sends the address to the fire department. The watch dog class checks whether the alarm and communicate channel are functioning properly. The fire department is informed about the address where the fire is.
All the classes in this example show state behavior and such classes are called reactive because they react to events. The state diagrams for the alarm class, reset class, communicate class and watchdog class are shown in figures 5-A, 5-B, 5-C and 5-D below.

**Figure 5-A state diagram**

![Figure 5-A state diagram](image)

**Figure 5-B state diagram**

![Figure 5-B state diagram](image)

**Figure 5-C state diagram**

![Figure 5-C state diagram](image)
The operation of each state diagram directly follows from the behavior of their respective classes. The watchdog class basically checks the channel alert and if there is an error which is given by the value of check==$0$ then it calls the system resetblink operation.

The modeling of real-time systems can be more systematic by using design patterns. Design patterns are a means of capturing the best design practices in the form of a design approach to a generic problem. They are often customized and tailored to meet specific needs of the application, just as arches in architectural systems are customized to meet the specific requirements of the building under construction.

The UML provides a notation for design patterns but this notation is targeted primarily towards mechanistic design patterns which deal with the construction of mechanisms – groups of objects that collaborate together for medium-sized purposes. Nevertheless, the notational schema also works for architectural design, which includes the placement of software modules on different processors, real-time scheduling policies, identification of concurrency models, and inter-processor and inter-thread communication.

The UML pattern notation is based on the UML class diagram. The elements of most interest are: (ref: http://www-md.e-technik.uni-rostock.de/ma/gol/ilogix/rtpatterns.pdf)

- Pattern (shown as a named oval)
- Class (shown as a named rectangle)
- Object (shown as a named rectangle in which the name is underlined)
- Package (shown as a tabbed folder)
- Relation
- Association (shown as a line connecting classes or objects)
- Aggregation (“has-a” – shown as an association with a diamond at the owner-end)
- Composition (a strong form of aggregation – shown with containment of classes within classes or with filled diamonds)
- Generalization (shown as an open arrow pointing to the more general class)
- Refinement (e.g. instantiation of templates into classes)
The issue to be taken into primary consideration in the following example is the safety and reliability of the smoke detector system present in the home. Safety and reliability category falls under architectural design patterns and includes the watch-dog design pattern as one of its variants. Safety and reliability must be provided by the combination of mechanical, electronic and software components.

Figure 7 watchdog design pattern
A simple solution for the time base problem is to use a watch-dog pattern, as shown in 
figure 7 above(ref: http://www-md.e-technik.uni-rostock.de/ma/gol/ilogix/rtpatterns.pdf).
In the watchdog pattern, an object called a watchdog is signaled periodically by various
objects in the system. The watch-dog pattern uses a hardware assistant called a watchdog
that can be informed in two ways with in a certain time frame that the system is
functioning properly. The watchdog basically observes both the alarm and the
communicate channel for every sixty seconds hence making it timer based where a timer
generates a signal every sixty seconds and the watch dog first pings the communicate
channel so that it reruns the IP address of the fire department, if it does then it gives the
value one to communicate channel. In the same way it checks whether there is sufficient
current in the ringing alarm of the bell, if it does then it gives that a value of one. Then it
AND’s both the values and passes the result to the watchdog. The watch dog checks if
the value is 1 or 0, if it is 0 then it calls the operating system to blink the errorlight and
reset the entire system as shown in the figure 8 above

The deployment diagram depicts the configuration of run-time processing nodes and
the components, processes, and objects that live on them. Software component instances
represent run-time manifestations of code units. Components that do not exist as run-time
entities do not appear on these diagrams. The deployment diagram for the smoke detector
system is modeled in the figure 9 below. It contains the components for the alarm class,
reset class, watch-dog class and communicate class deployed on the processor node.

Figure 8 watchdog operation
An object diagram is one that encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a collaboration diagram. The Object Model describing the behavior of various objects belonging to the respective classes of the smoke detector system is shown in Figure 10 below.

**Figure 10 Object diagram**

- **Watchdog class**
  - `check=1` //is memory mapped to an area which takes the ANDed value from the alarm and the communicate channel
  - `checkchannel()`:void
    - //compares the value of `check` and if zero calls the system `resetblink()` function

- **Alarm class**
  - `readvalue=10` //is memory mapped to an area which takes the value from `lightsource()`
  - `avgvalue=7`
  - `soundalarm()`:void
    - //calls the system `ringalarm()` and then either calls the reset or communicate class

- **Reset class**
  - `takevalue=0` //is memory mapped which takes the return value of `pressed()`
  - `count=50` //time to wait
  - `disablealarm()`:void
    - //calls the system `sleepalarm()` based on the `takevalue`

- **Communicate class**
  - `address= “363 N, manhattan, KS”`
    - //address to send
  - `send()`:void
    - //calls the system communication to send the home owners address to the fire department

- **Operating system class**
  - `lightsource()`://writes the value of light intensity to a memory location
  - `resetblink()`://resets the entire system
  - `channelalert()`://writes value 1 or 0 based on whether the alarm and communicate are working to memory
  - `ringalarm()`://horns the alarm to the home owner
  - `sleepalarm()`://disables the alarm for a specific period
  - `pressed()`://writes 0 or 1 to a memory location based on if the reset is pressed or not
  - `communication(address)`: //sends the home owners address to the fire department through the communication channel
Finally the real-time model can be extended using stereotypes to represent the structure as a concurrent system where several threads of various objects present will be executing in a parallel fashion. The UML uses the <active> object stereotype to represent an object that is the root of a thread. The <active> stereotyped class is typically a composite class that tightly aggregates its components. The <active> object creates and destroys those components, and the components executing within the thread of their composite <active> object. We can graphically depict these objects on a task diagram shown in figure 10 below, which is the usage of a UML object diagram on which only <active> objects are shown. UML along with its extensions provide us with a reliable and manageable architecture for Safety Critical Real Time Systems.

**Figure 11 Task Diagram**

The task diagram of the smoke detector system shown in figure 11 above contains threads for executing various tasks. The software for the system can be written in real-time operating systems integrated development environment, example Tornado(VxWorks) or LeJos(LeJos). The kernel part of the software in the system will contain various threads for message handling and task execution.

For example when the watch dog thread is active and calls the operating system thread to find out the status of the communicate and alarm channels, the OS thread makes a task switch and examines the system thread for the necessary parameters and then reverts back to the older watch dog task(another task switch). Also when the communicate thread is executing and calls the hard wired communicating(address) function, there must be a message switch that must handle the passing of string address from the communicate thread to communicating function which is executed by the message handling thread of the kernel. Hence we have several active threads of the application part and some active threads of the kernel part.
4. Source code in C++

The source code for the above classes can be written using an object oriented programming language such as C++ or java and is shown below.

//implementation of the System

//header files dependent on the operating system being used
#include <task.h>
#include <msglib.h>
#include <conio.h>
#include <string.h>
#include <iostream.h>

//classes will be called by different names in different environments
class communicate
{
    char address[50]; //the address of the home owner

public:
    void send(char*);
} com;
communicate::send (char location[])
{
    strcpy(address,location);
    communication(address);//system built function that sends the address of the home owner to the fire department through a communication channel and informs the watch dog that the alarm is working properly
}

class reset
{
    bool takevalue;//whether the reset button is pressed or not
    int count;//wait for some time for the home owner to press the reset button

public:
    void disablealarm( );
} button;
reset::disablealarm()
{
    count = 50;
    while(count !=0)
    {
        takevalue = pressed( );//system built function that takes the value of the reset button present on the smoke detector system
    }
if (takevalue == 1) // if reset button is pressed
{
    sleepalarm(); // system built function that disables the alarm for a period of time and then
    // activates it
    exit(); // exit the function after sleepalarm is finished with its task
}
count--; 
}
com.send("363 N, 14th st, Manhattan, KS");
}
class alarm
{
    int readvalue; // to read the value from the light source
    float avgvalue; // to compute the average of the smoke density
    public:
        void soundalarm();
    } horn;
alarm::soundalarm()
{
    readvalue = 0;
    avgvalue = 0;

    for(int i = 0; i < 100; i++)
    {
        readvalue = lightsource(); // system built function that takes the reading of the light
        // falling on the alarm sensor
        avgvalue = avgvalue + readvalue;
    }
    avgvalue = avgvalue / 100; // density of the smoke for 100 readings

    if (avgvalue > 5 && avgvalue < 10)
    {
        ringalarm(); // system built function that rings the alarm and informs the watch dog that
        // the alarm is working properly
        button.diablealarm(); // if the smoke density is less check whether the user wants to stop
        // the alarm, if not then communicate the fire department
    }
    else if (avgvalue > 10)
    {
        ringalarm();
        com.send("363 N, 14th st, Manhattan, KS");
    }
void main()
{
  while(1)
  {
    horn.soundalarm();
  }
}

//implementation of the Watchdog

//header files dependent on the operating system being used
#include<task.h>
#include<msglib.h>
#include<conio.h>
#include<string.h>
#include<iostream.h>

//classes will be called by different names in different environments

class watchdog
{
  int check;//to check the functioning of communicate channel and alarm

public:
  void checkchannel( );
} watch;

watchdog::checkchannel()
{
  check=0;

  check = channelalert();//system dependent function which ANDS both the alarm and communicate channel values and returns 1 if active otherwise returns 0;

  if(check==0)
  {
    resetblink();//system dependent function that blinks a light saying that some channel is not functioning properly and then resets the entire system
  }
}

void main()
{
  int count==60;//check the channels every count seconds

while(1) {
    if(count==0) {
        watch.checkchannel();
        count=60;
    }
    count--;
}

**Conclusion:** This paper explains the importance of UML in the architectural design of real time systems, thus we can model a fault tolerant or a safety critical real-time system using the various features of UML and its extensions through stereotypes. Since the use of UML for modeling real time systems is in its basic stage, this paper is open for further discussions and updates. This model can also be potentially reused for other systems that exhibit similar functionality as it represents the characteristics of both time driven and event driven systems.

**Acknowledgement**

I would like to thank Dr Bill Hankley for his patience, time and valuable feedback throughout the compilation of the paper.

**References:**

