Designing a Home Alarm using the UML

And implementing it using C++ and VxWorks

M.W.Richardson I-Logix UK Ltd. markr@ilogix.com

This article describes how a simple home alarm can be designed using the UML (Unified Modeling Language) and targeting the VxWorks OS. The article is in two parts, the first describes how a model of the home alarm can be designed and validated using the UML, so that it is independent of the actual hardware and OS used. The second part details how this model can then be specialized to use the VxWorks operating system running on a 486 target. The design described is modeled, developed and validated using the Visual Programming environment "Rhapsody" from I-Logix.

Being that the UML is a very visual Programming Language, this article is also very visual.

Part I: Designing a HomeAlarm that is OS independent

A good Object Oriented design tries to separate the things that change from the things that don't change. In a home alarm, one thing that is likely to change, is the actual hardware platform. This article describes how the model can be designed in UML so that different hardware architectures can be easily accommodated. This article also shows how by using "*Rhapsody*", full production quality code can be generated from the model and also validated by using "Design level debugging". Design level Debugging will allow us to debug using the very same diagrams that we have used to describe the model, basically the Statecharts and Message Sequence Diagrams will come to life or "animate". This will allow us to rapidly validate the model.

The Requirements

We want to design an alarm system for a home that has the following requirements:

Can be armed / disarmed either via a remote control unit or via a simple keypad. When using a keypad to arm the alarm, a four-digit code must be entered followed by the on key. To disarm the alarm, a four-digit code must be entered followed by the off key. It should be possible to change the code. When the alarm is armed and a presence is detected, the alarm will sound immediately. When arming the alarm, there is an exit delay to allow the homeowner to exit. When opening the door when the alarm is armed, there is an entry delay during which the homeowner can disarm the alarm, failing to do so will result in the alarm going off.

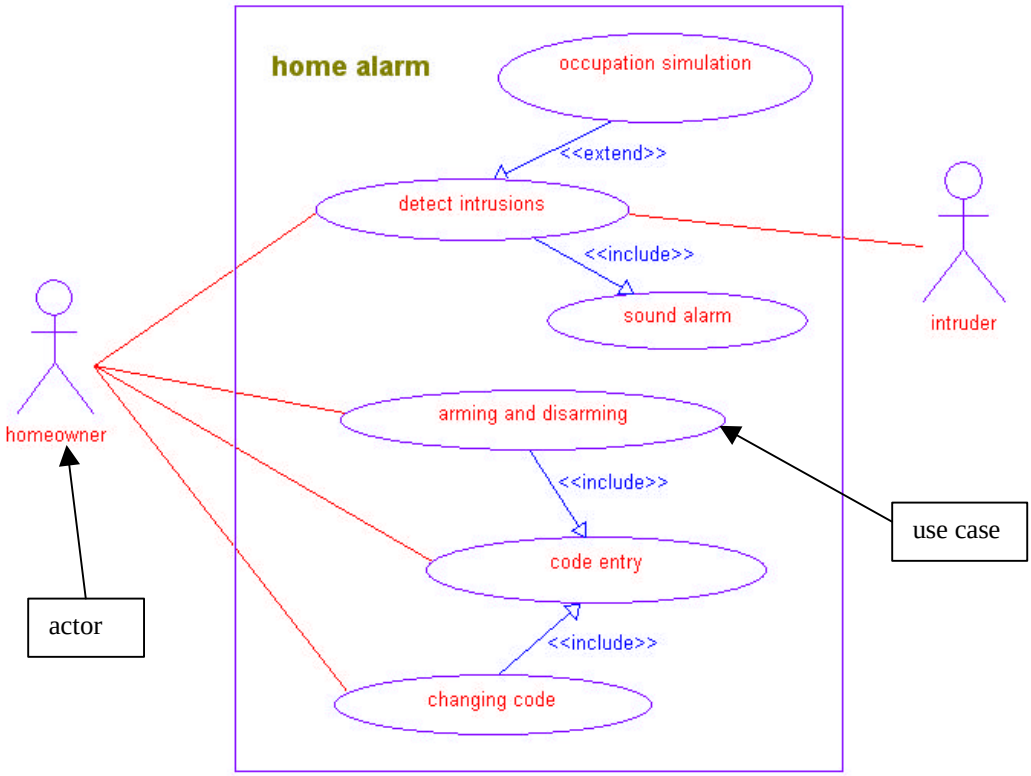
The alarm will have two LED's, one green and one red. The red LED will indicate the alarm is active, and will flash either when the alarm is being activated or if the alarm is activated and the door is opened. The green LED will indicate that the alarm is powered up and will flash four times when the code has been correctly changed.

A future requirement is that the home alarm system should also be capable of switching lights on/off to simulate presence in the house.

Designing the UML model

Use Case Diagram

The first step is to build a use case diagram from the requirements. The following Use case diagram, shows two things: Firstly it shows the main uses (or use cases) of what we are trying to model “Home Alarm”. Secondly it shows who uses the home alarm; we have two principal actors or users, the homeowner and an intruder. A use case is a system function that returns an observable result to an actor without revealing internal system structure.

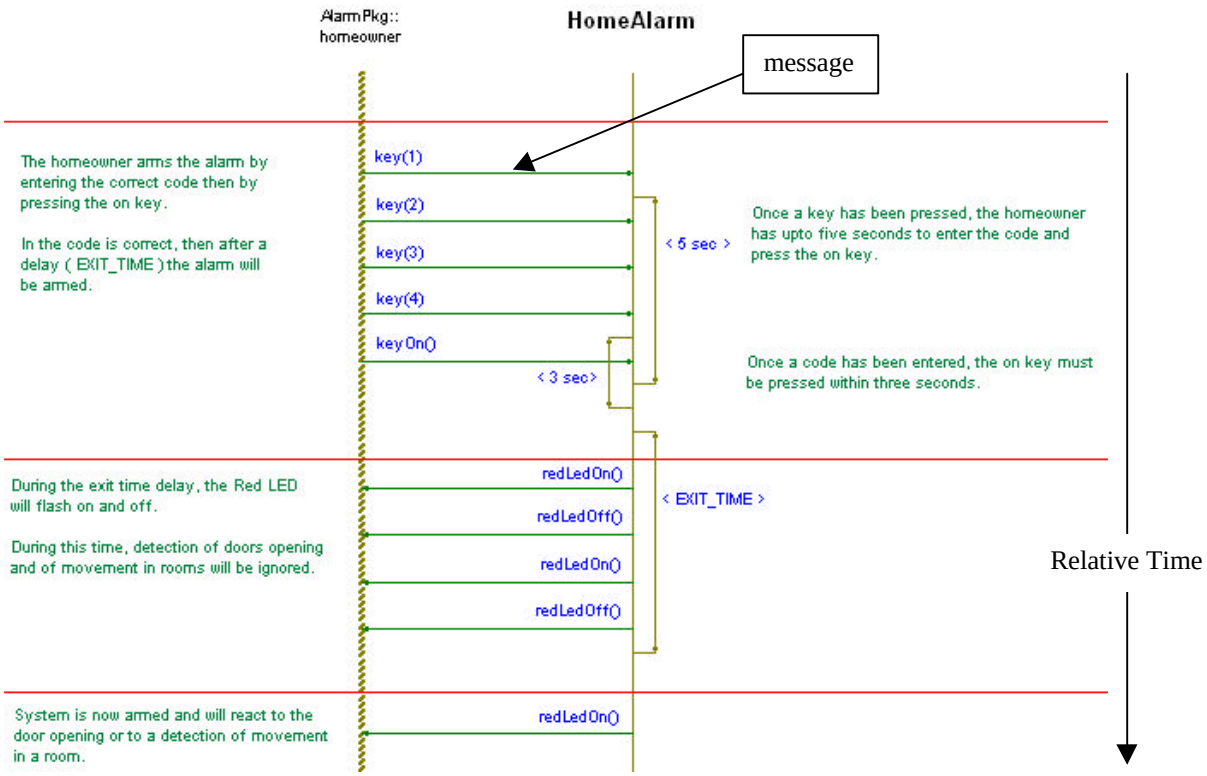


In the above diagram we can see that the principal use of the home alarm is the use case “detect intrusions”, in which both the homeowner and the intruder participate. The use case “occupation simulation” extends “detect intrusions” by effectively switching lights on and off to simulate occupation when the alarm is armed. Two other use cases that the homeowner participates in are “arming and disarming” and “changing code”; both of which use or include the services of another use case “code entry”. For each use case we can add a textual description and if needed, the use case can be decomposed into further use cases on another diagram.

The use cases do not imply any internal structure and are a functional view.

Black Box Scenarios

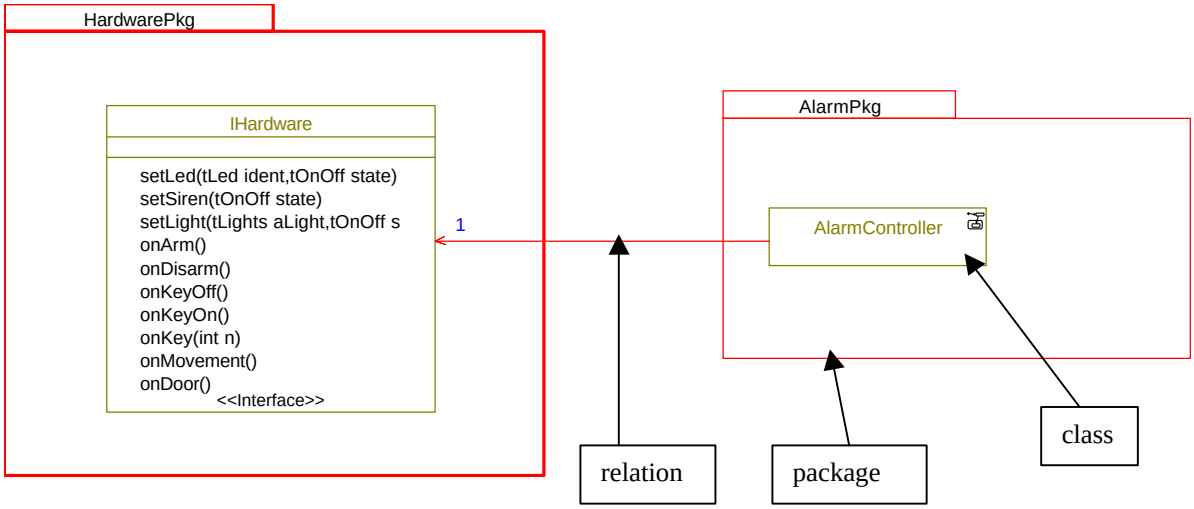
Having created a use case diagram, the next step is to look at some black box scenarios. The idea of these scenarios is to show the interaction between the system that we are trying to model, and the actors. We regard the system as being a black box. For example: for the use case “Arming and disarming” we could imagine the following scenario:



The above scenario has been captured using a Message Sequence Diagram and shows the interaction between the homeowner and the home alarm over time. When creating such diagrams, many more details about the system are discovered. For example in the above scenario, timing information has needed to be defined between key presses.
These scenarios help us get a better understanding of how the users use the system and also help us understand exactly what needs to be done.
Generally for each use case there would be several interesting scenarios that need to be captured. Each scenario is effectively an instance of a use case.

Domain Diagram

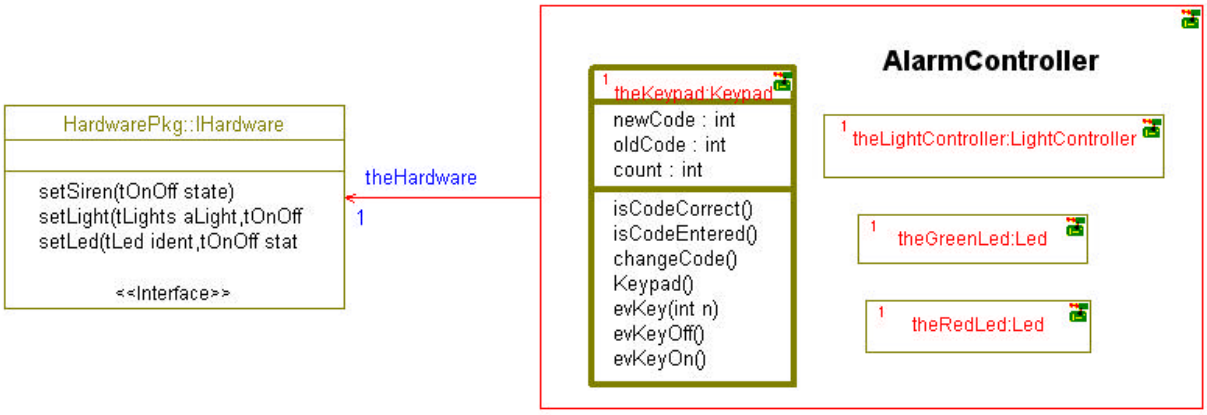
The next step is to see how we can break the model down into smaller sub-systems or domains. A domain can be shown in the UML as a package, which is basically a collection of classes. The following domain diagram shows how the home alarm has been divided into two basic packages; an alarm domain (AlarmPkg) and a hardware domain (HardwarePkg). The idea here being that the alarm domain is completely independent of the actual hardware and any hardware domain that realizes the *IHardware* interface can be used in conjunction with the alarm domain. The interface class *IHardware* describes all the operations that are necessary for the alarm domain. The *set* operations are operations that the alarm domain can invoke on the hardware, whereas the *on* operations can be invoked by the hardware.



Once this interface has been defined between the alarm domain and the hardware domain, more detailed analysis can now be undertaken simultaneously and independently on each relevant domain, each domain knowing the interface between itself and the other domain(s). For example we can now start to analyze the alarm domain without knowing anything more about the hardware that will be used.

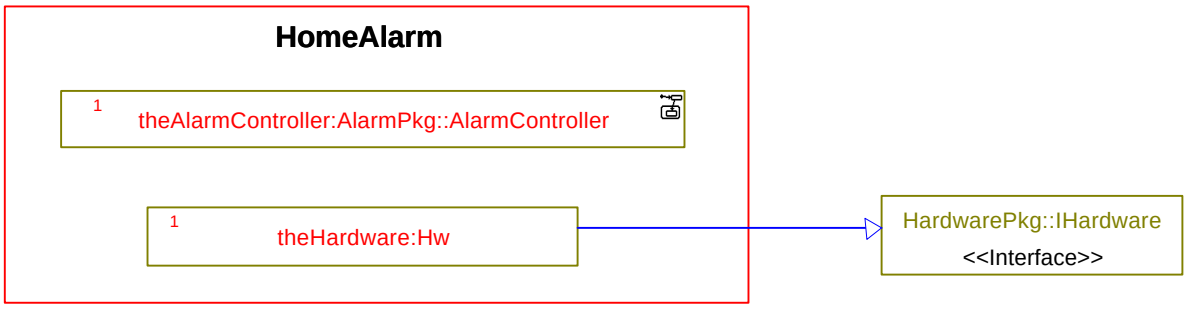
Object Model Diagrams

For the alarm domain (AlarmPkg), we can imagine that we have an AlarmController class that contains Keypad, Led and LightController classes. These classes can be shown as follows:



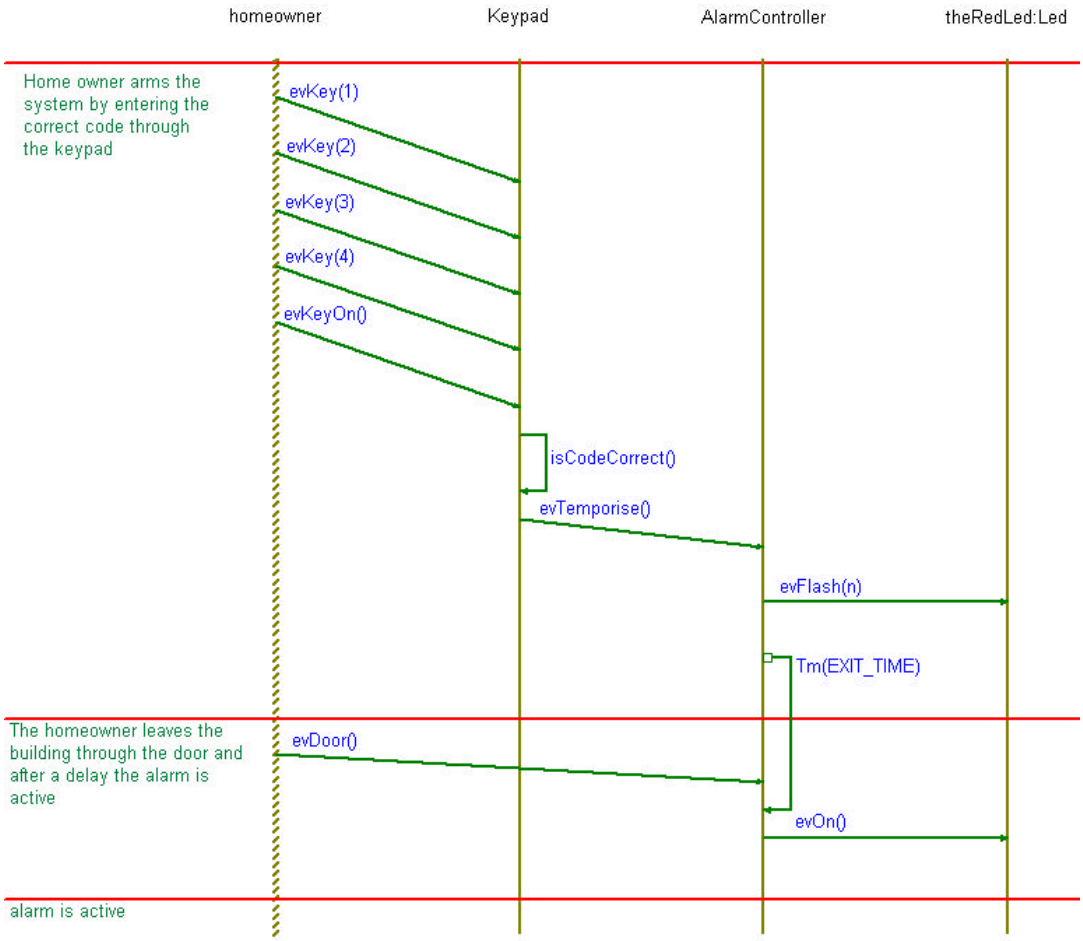
This diagram shows the AlarmController as a composite class containing instances of the other classes. We can also see that we have chosen to display various operations and attributes of the Keypad class. A composite class is an effective way of showing strong aggregation. In the above example if an instance is created of the AlarmController class, then it will create an instance of the Keypad called *theKeypad*, two instances of the Led class called *theRedLed* and *theGreenLed* as well as one instance of the LightController class called *theLightController*. If the instance of the AlarmController is deleted then the contained instances will also be deleted.

The IHardware class has some pure virtual operations, so in order to test the AlarmController, we must override these operations. The following diagram shows that a HomeAlarm contains one instance of the AlarmController class and one instance of a Hw class that inherits from and overrides operations in the IHardware class.



White box Scenarios

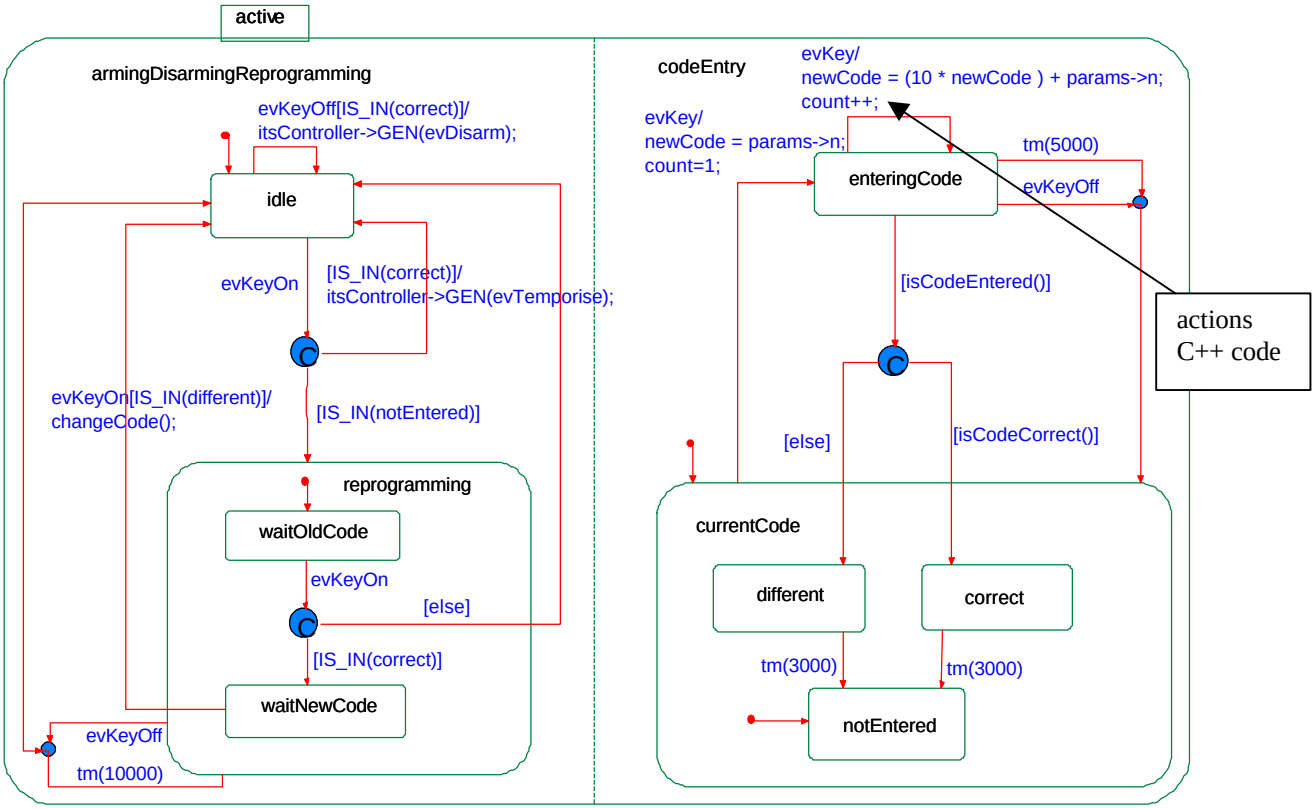
Now that we have found a few classes, we can start to expand on each of the black box scenarios to produce white box scenarios. Again Message Sequence Diagrams will be used this time to capture the interaction between instances of the above classes during various scenarios. For example the following sequence diagram describes the scenario where the homeowner arms the alarm and leaves the house.



The inclination of the messages indicates whether or not the message is synchronous (horizontal) or asynchronous (diagonal). From the above diagram we can see that many of these classes have behavior, they are handling timeouts, calling operations on other classes and also receiving events. In UML, this behavior can be specified with a statechart.

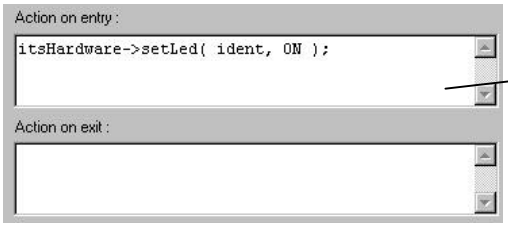
Statechart of Keypad

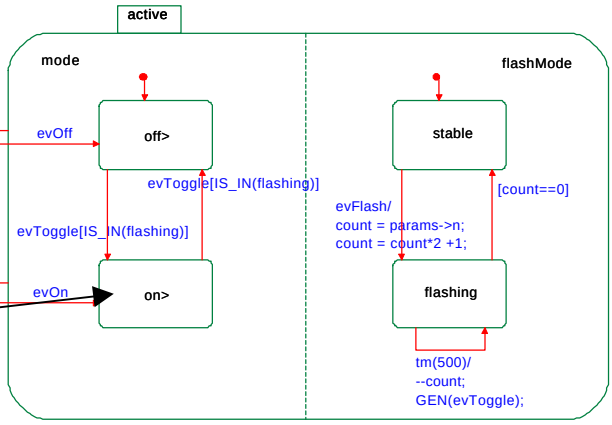
The following statechart describes the behavior of the keypad: Often when the problem is complex, the problem is divided into several simpler problems. This is exactly what has been done here with the keypad. Concurrent states have been used so that the "code entry" can be handled with one concurrent state and the higher level functions of "arming, disarming and reprogramming" can be handled with another concurrent state. We can see that on the statechart we can enter as actions C++ code, we can also call various operations such as *isCodeEntered()* , *changeCode()*, … We are also using various attributes such as *count* and *newCode*.



Statechart of LED

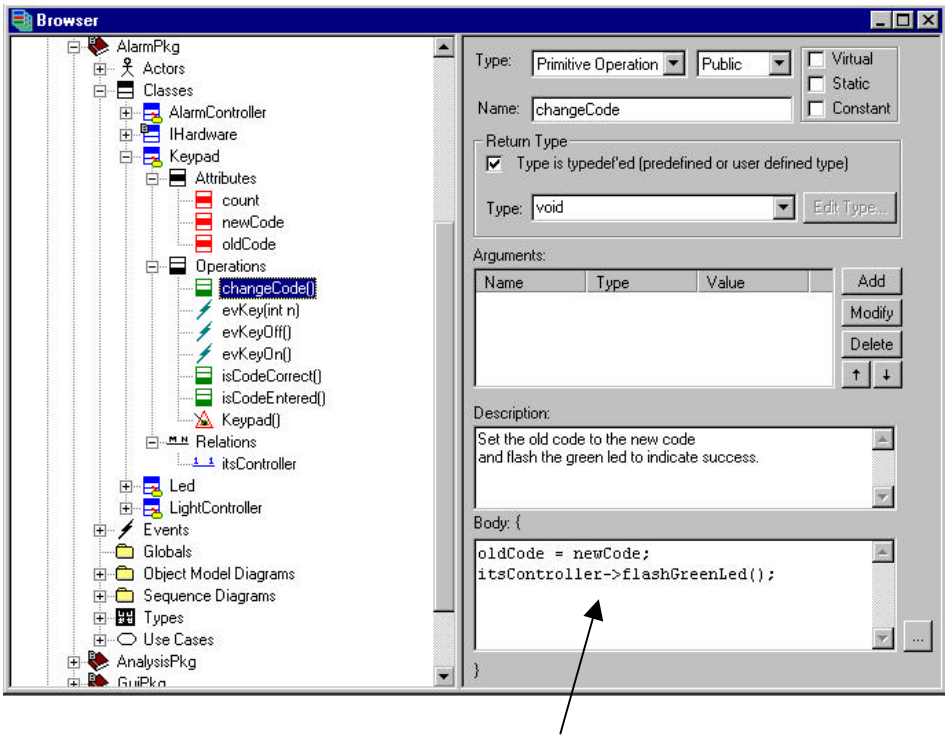
The following statechart describes the behavior of the LED class. Again, in this statechart composite states have been used. In the on and off states, the symbol > indicates that there are entry/exit actions. For example in the on state there are the following actions:





Operations and Attributes

We have seen that many classes have attributes and operations, these can be specified as follows, note that the actual C++ code is contained in the model.

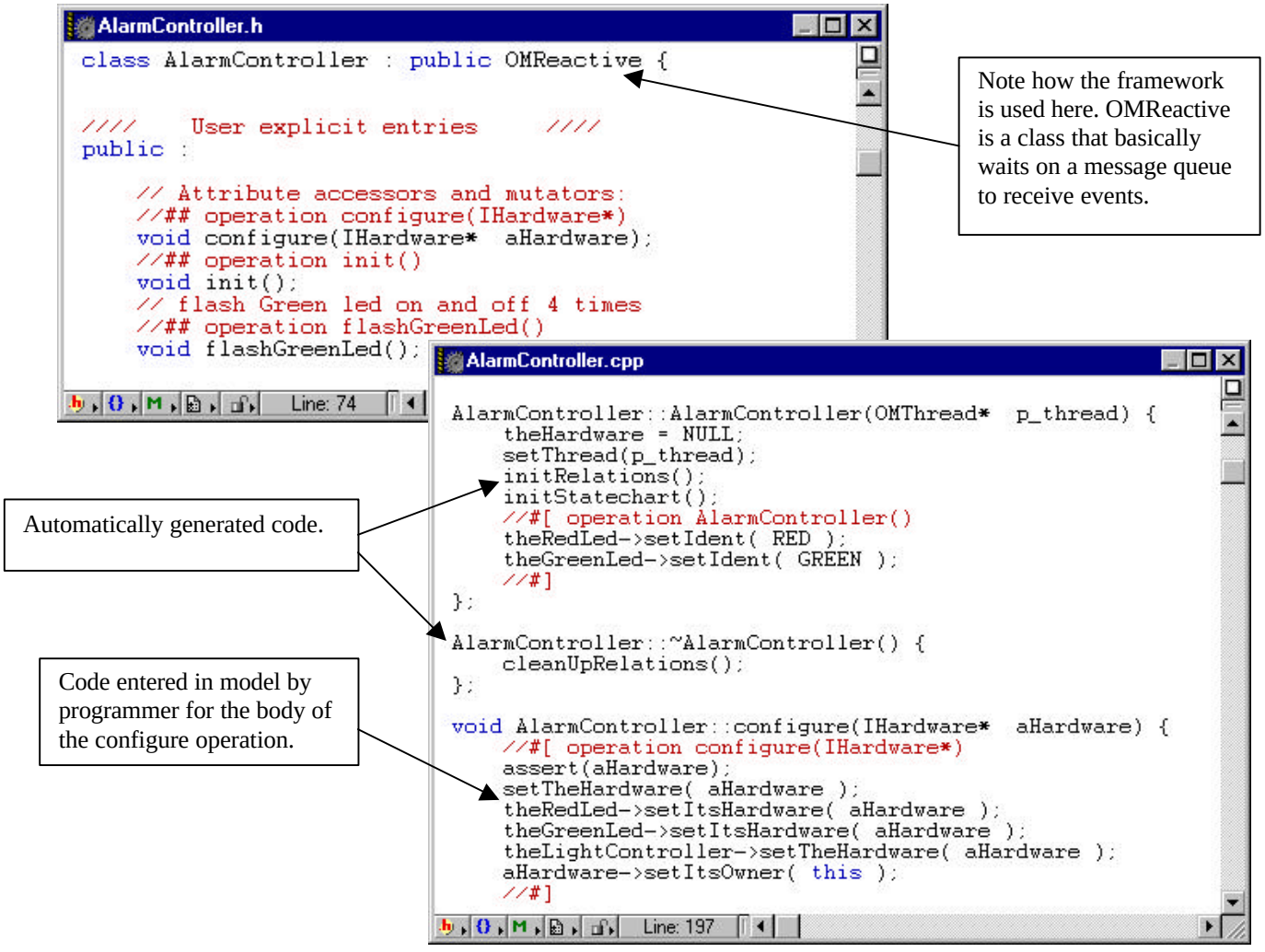


A tool such as Rhapsody can generate code automatically from a UML model, but of course often some lines of code have to be written by hand such as class operations and statechart actions, this can be done within the tool as shown above.

Generating code

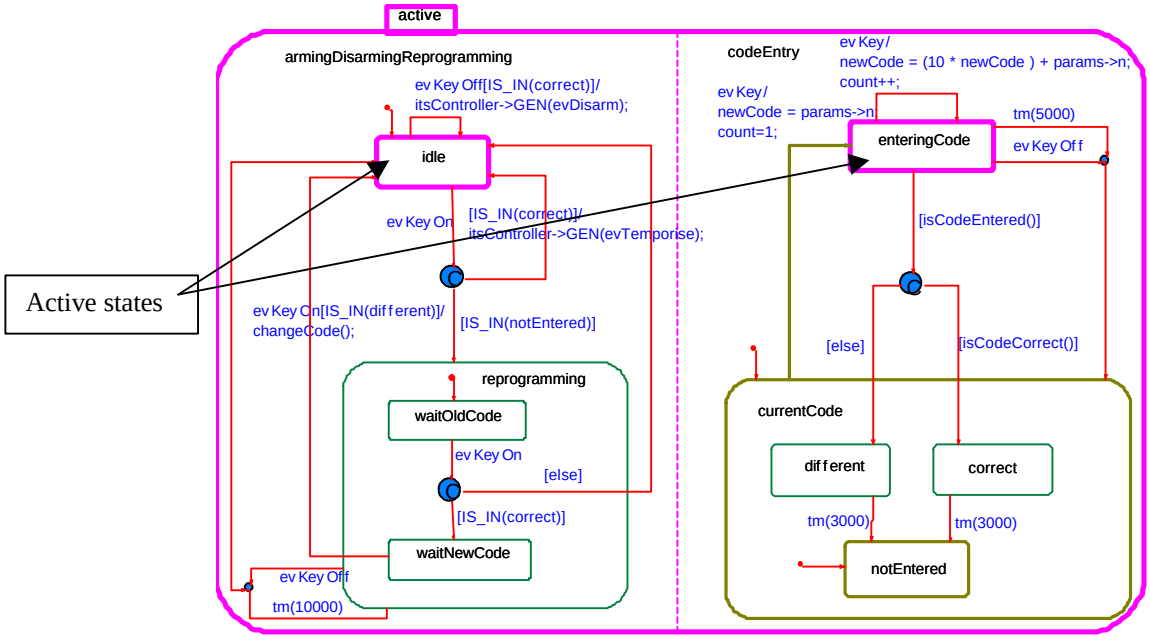
The UML is rigorously enough defined to permit automatic generation of code from the Statecharts and the Object Model Diagrams. Every time when doing an Object Oriented project, some kind of framework has to be created. This framework will perhaps implement a mechanism for handling statecharts, perhaps a mechanism for interfacing to an OS. It will also describe how to implement relations, how to communicate between classes on different threads, how to handle timeouts, … This framework can provide around 60 to 90% of the total code required to implement the application. Imagine writing a Windows program without using the MFC framework, Rhapsody provides a framework (designed and optimized for embedded real-time projects) which allows the embedded real-time programmer to concentrate on the application and not on the low-level implementation. When such a framework is used, automatic code generation becomes efficient and relatively straightforward. The Rhapsody framework also contains an abstract operating system interface allowing the model to be independent of the actual OS used.

Once the remaining diagrams have been drawn and the bodies of various operations completed, we can proceed to generate code and test the model. Within Rhapsody we need to setup a component and configuration to tell Rhapsody what classes to generate code from and what environment to use, normally to start with, we use the Microsoft environment (Windows OS and Visual C++ compiler). Then code can be generated and compiled within Rhapsody to generate an executable. The following is an extract of the code generated for the AlarmController class:



Validating the UML model

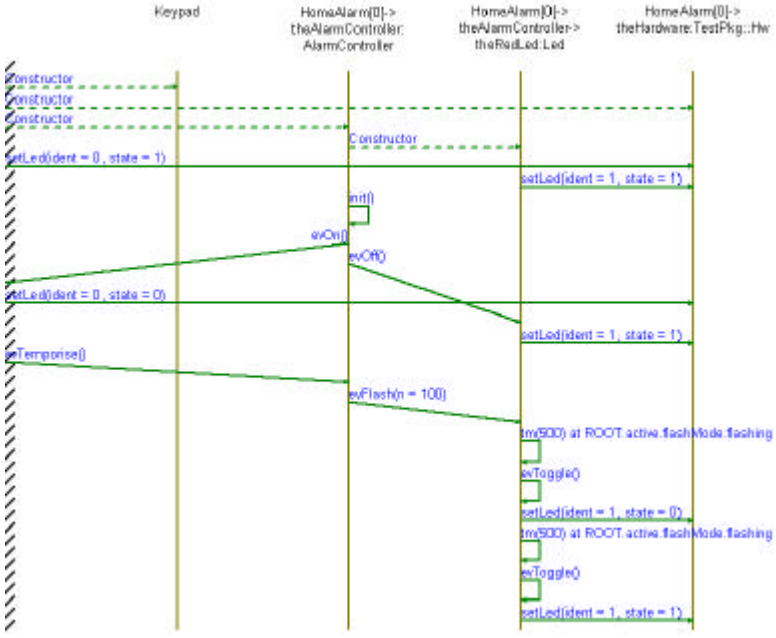
Rhapsody can instrument the generated code so that when the actual code is executed it feeds information back to the tool, allowing Rhapsody to animate the model. This allows the programmer to debug at the design-level. This can be done at any time during the development, even from day one without a single line of code being written by the programmer. Design-Level debugging allows problems to be detected very early on in the design cycle and put right immediately, thus resulting in a faster time to market.



In the above diagram we can see an animated statechart of an instance of the Keypad class showing that this instance is in the *idle* state and the *enteringCode* state. The active states are highlighted.

Events can be manually injected, and animated sequence diagrams such as this one, can be used to automatically capture the interaction between instances as they occur.

This animated sequence diagram can then be examined or even compared to a sequence diagram drawn by hand during analysis to ensure that the model is correct. This comparison, can of course be done automatically by Rhapsody.



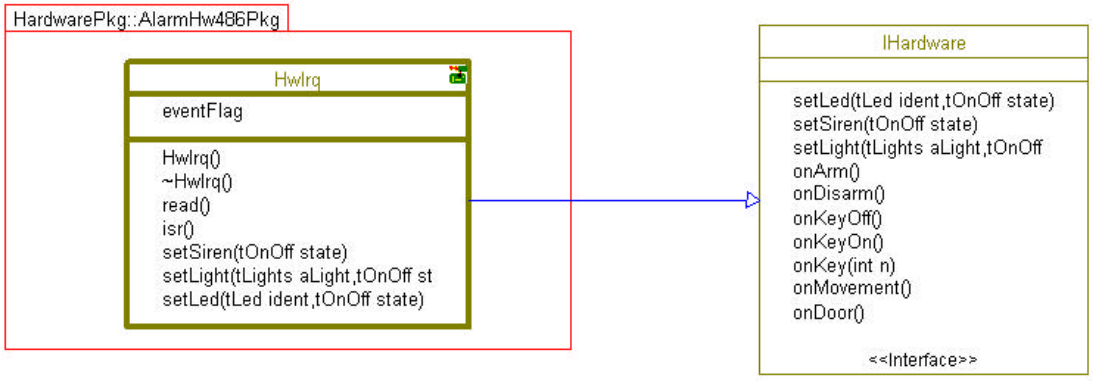
Testing the UML model with a simple GUI

Once the Alarm domain has been tested by manually injecting events and observing what happens, a simple GUI (Graphical User Interface) can be created using for example Visual C++ from Microsoft. The class used to create this GUI could inherit from the IHardware class, specializing the ”*set*” operations, and calling the “*on*”operations whenever a button is pressed. The GUI can then be used to drive the model that can again be debugged at the design-level.

Using a GUI like this also enables both Customers and Marketing to verify that the product behaves as they expected. Any necessary modifications can be easily made to the model and a new executable generated with a simple press of a button.

Part II: Targeting VxWorks

Once the home alarm has been designed and validated, it can then be specialized to target the *VxWorks* operating system running on say a ns486 board to which a keypad, two LED's, a siren, a movement detector and a door switch are connected via an I/O board. The hardware is such that the keypad, movement detector and door switch all generate an interrupt.



A new class called HwIrq can be created that inherits from the interface class IHardware and that specializes the "set" operations. The body of these operations can be written within Rhapsody. To write to the I/O board, the VxWorks operation *sysOutByte()* has been used.

HwIrq.cpp

```
OMOSEventFlag*  HwIrq::eventFlag = NULL;
HwIrq::HwIrq(OMThread*  p_thread) : portA(0xff),portB(0),
    setThread(this, TRUE);
    initStatechart();
    //#[ operation HwIrq()
    #define INPUTS_INT_NUM  (INT_NUM_IRQ0 + 3)
    #define INPUTS_INT_VEC  (INUM_TO_IVEC(INPUTS_INT_NUM)

    eventFlag = theOSFactory()->createOMOSEventFlag();
    int x = intLock();
    intConnect(INPUTS_INT_VEC, (VOIDFUNCPTR)(isr), 0);
    intUnlock(x);

    sysOutByte (PORT_CONTROL, 0x8B);
    sysOutByte (PORT_A, portA);
    sysOutByte (PORT2_CONTROL, 0x8B);
    //#]
};

void HwIrq::setSiren(tOnOff  state) {
    //#[ operation setSiren(tOnOff)
    if ( state == ON )
       portA &= ~SIREN;
    else
       portA |= SIREN;
    sysOutByte (PORT_A, portA);
    //#]
};

void HwIrq::isr() {
    //#[ operation isr()
    eventFlag->signal();
    //#]
};

void HwIrq::read() {
    //#[ operation read()
    while ( 1 ) {
        eventFlag->wait();

        // read all inputs
        int newPortB = 0xff - sysInByte (PORT_B);
        int newPortC = 0xff - sysInByte (PORT_C);

        if ( newPortB != portB ) {
            if (( portB & KEY_0 ) == KEY_0 ) onKey(0);
            if (( portB & KEY_1 ) == KEY_1 ) onKey(1);
```

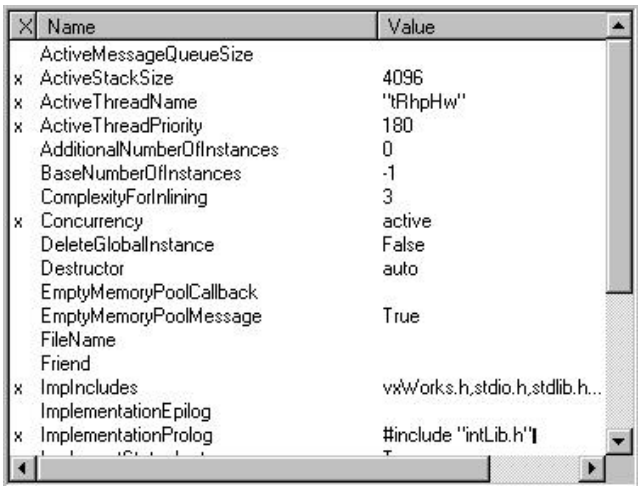
Line: 80

The constructor sets up one port as an output and two ports as inputs. It also uses the VxWorks operation *intConnect()* so that the routine isr() is called whenever IRQ3 occurs. Finally it creates an event flag that will be signaled from the isr routine.
The isr() routine is a static operation that simply signals the eventFlag on which the read operation is waiting. This read operation uses another VxWorks operation *sysInByte()* to read the inputs.

Within Rhapsody we need to setup a new component and configuration to tell Rhapsody what classes to generate code from and what environment to use, in this case, we use the VxWorks486 environment, (VxWorks OS for the 486 and the gnu compiler).
Then code can be generated and compiled within Rhapsody to generate an executable that can be downloaded to the Tornado debugger and executed.

If the code was generated with "Instrumentation" then Rhapsody will allow the model to be debugged at the design level in exactly the same way as described previously. Tornado can be used to debug at the source level at the same time as Rhapsody debugs the model at the design-level.

The HwIrq class has been set as an *active* class and so will run on its own thread, the parameters for this thread can be easily configured within Rhapsody as shown to set the thread name to "*tRhpHw*", the stack size to *4096* bytes and the priority to *180*.



Finally, code can be generated from the model with no instrumentation that can then be debugged in the classic way using Tornado and perhaps WindView to check system performance before delivery. Documentation can be also generated from the model with a single button press, after all who likes to spend time (or indeed has time) to document a project.

Conclusion

This article has shown the following:

1. How Rhapsody can be used to model using the UML
2. How the model can be debugged at the design-level.
3. How the model can be debugged with an external GUI.
4. How the model can be targeted to a specific hardware platform running VxWorks.

References

	Author	*Reference*
1	***Bruce Powel Douglass***	***"Doing Hard Time" ISBN 0-201-49837-5***
2	***Bruce Powel Douglass***	***"Real-Time UML" ISBN 0-201-32579-9***
3	***I-LOGIX Inc***	***http://www.ilogix.com***

Nohau Elektronik AB
Derbyvägen 4, SE-212 35 Malmö - SWEDEN
Int tel: +46(0)40 59 22 00, Int fax: +46 (0)40 59 22 29, www.nohau.se

Nohau Danmark A/S
Naverland 2, DK-2600 Glostrup - DENMARK
Int tel: +45 43 46 63 93, Int fax: +45 43 46 63 94, www.nohau.dk