ASPECT-ORIENTED SOLUTION FOR MUTUAL EXCLUSION IN EMBEDDED SYSTEMS

A. Gomola*

[*]Institute of Automotive Mechatronics,
Faculty of Electrical Engineering and Information Technology,
Slovak University of Technology in Bratislava,
Ilkovičova 3, 812 19 Bratislava, Slovak Republic
(e-mail: alojz.gomola@stuba.sk)

Abstract: Embedded systems are developed for wide range of applications. The best known applications are industrial process control and banking solutions. Fault tolerance is the crucial requirement in long-term embedded systems. This paper presents solution for mutual exclusion in embedded systems. The usual mutual exclusion solution using semaphores is a crosscutting concern. Semaphores are difficult to maintain in code and their failure rate is high. We propose new aspect-oriented solution for mutual exclusion. Our solution utilizes aspect-oriented approach, is usable in other systems and designed to be robust against program changes, and it provides a solution to aspect fragility problem.

Keywords: Embedded systems, Aspect oriented programming, Robust algorithm, Mutual exclusions, AspectC

1 INTRODUCTION

From the spread of object-oriented paradigm the need for architectural and abstract approach to embedded systems software design arose. This need was satisfied through publication of work that brought the concept of layered system (Kopetz, 1997). The author divided the system into several layers and specified required functionality for each layer:

1. Hardware layer,
2. Resource management layer,
3. Service layer,
4. User interface layer.

Hardware layer creates wrappers for direct access to peripheral devices and operating system resources. *Resource management* layer processes the outputs of peripheral devices and is responsible for protection of shared resources within the system. In distributed systems it uses interface to conceal the distributed nature of resources. *Service layer* implements the logic of the controlled process. It abstracts the real world process that is controlled by the embedded system, and it implements individual transactions that the embedded system performs. *User interface* layer is an optional layer that is present only in systems that communicate directly with the user (e.g., POS terminal). It influences all the other above mentioned layers (Baleani *et al*., 2005).

Advanced and distributed embedded systems have their own operating system. This operating system is based on UNIX-like operating systems used on personal computers and servers. Examples of operating systems for embedded systems are *Minix* and *EmbeddedBSD* . Great disadvantage of these systems is that they offer only very limited functionality as far as the use of peripheral devices (Deng *et al*., 2006). operating memory (Hoffer *et al*., 2007) and threads is concerned. The solution of this problem was found in aspect-oriented approach. In the works of German team that was developing embedded systems for European space program, following problem areas (Alsonso *et al*., 2007, 2008), in which aspects provide a solution, were identified:

1. Fault tolerance (Alsonso *et al*., 2008),
2. Middleware customization (Alsonso *et al*., 2007),
3. Platform variability (Alsonso *et al*., 2007),
4. Transaction management (Alsonso *et al*., 2008),
5. Synchronization, mutual exclusion (Alsonso *et al*., 2008),
6. Logging in a real-time embedded system (Alsonso *et al*., 2007).

1.1 Fault tolerance

Even though *fault tolerance* is not a domain specific problem of embedded systems, it is the most closely related one to them. Embedded systems have to implement maximum *fault tolerance*, as some of them have to operate ceaselessly for decades. System faults are caused by *incorrect use of peripheral device* (Alsonso *et al*., 2008), *faulty operating memory* management (Hoffer *et al*., 2007), *memory overwriting* (Hoffer *et al*., 2007). The solution for mutual exclusion using thread locks was proposed in the article (Alsonso *et al*., 2007)]. We do not consider this solution to be sufficient because it defines overcomplicated pointcuts and it would be necessary to modify the weaving with every change of the application. In our opinion, given approach completely bypasses the goal of aspect-oriented approach, which is to separate crosscutting concerns into the robust aspects. The authors (Alsonso *et al*., 2007) referred to the aspect for handling the semaphore faults, but this approach violates the conditions of mutual exclusion and can lead to system destabilization.

1.2 Synchronization

Synchronization can be intrasystem or intersystem. Intrasystem synchronization is handled by the system of mutual exclusion as far as the access to shared resources or peripheral devices is concerned. For this purpose we usually use semaphores or their variations (locks, memory access management). Intersystem synchronization (Baleani *et al*., 2005) serves as synchronization among embedded systems. There are two models of intersystem synchronization:

1. Competitive,
2. Controlled.

In *competitive synchronization* distributed embedded systems compete for available resources and are not in homogenous environment, i.e., one system sends a request to another system for processing, and this system serves the earliest request first. This concern is tangled in the transactional management of the request's target system.

In *controlled synchronization* there is a superordinate system that controls the access to shared resources and their usage. An example of controlled synchronization is the cooperation of ABS embedded systems, which are synchronized and controlled by the central ABS system, which tells them where to acquire the data for processing. The main embedded system interferes with this communication by keeping the bus free and limiting the access of other embedded systems, e.g., radio. Intersystem synchronization is a rather complicated problem that is related to complex distributed systems. It is the problem of distributed programming, and this concern would be very difficult to detach into aspects.

2 PRELIMINARIES AND PROBLEM FORMULATION

Aspects are weaved into existing code and the place, to which they are weaved, is defined in *pointcuts*. From changes to the source code, the need for changes in *aspect pointcuts* definitions arises. Generally, it would be correct to handle aspects and program separately. *Aspect pointcuts* should be *robust*, and there should be no need to change them. This is not always achievable because when aspect implements business logic, its binding always changes.

Mutual exclusion of processes or threads in embedded systems is related to two problem areas: 1. Synchronization, 2.Fault tolerance.

2.1 Promiscuous semaphore in the form of aspect

We designed a *robust solution for mutual exclusion* that is implementable as an abstract aspect, *allows multi-level function locks* and is robust as far as changes to source code are concerned. Given solution is broadly applicable and utilizes standard resources provided by the operating system of the embedded system. The extension uses the operating system method to determine the identification number of a process or a thread. The *identification number (ID*) is unique and serves for exact identification of semaphore owner.

```
Aspect Promiscuous_Mutex {
  private:
    Mutex *master, *slave;
    int owner_id;
    int counter;

  void create() { master = new Mutex;
    slave = new Mutex;
    owner_id = HW_MUTEX_NO_PROCESS_ID;
    counter = 0;}

  void dispose() {delete master; delete slave;}

  void acquire() {...}

  void release() {...}

  //pointcuts
  pointcut virtual
    binding_object_class() = NULL;
  pointcut virtual
    protected_functions() = NULL;

  //advices
  advice construction (binding_object_class())
      : before () { create(); }
  advice destruction (binding_object_class())
      : after ()  { dispose(); }
  advice execution (protected_functions())
      : before () { acquire(); }
  advice execution (protected_functions())
      : after ()  { release(); }
}
```

Code 1 Abstract aspect implementing the promiscuous semaphore.

The *Thread.get_current_id()* function is a wrapper function that returns the ID of caller thread or process. These functions are provided by the standard ISO C++, therefore, the presented solution is reusable.

The *promiscuous semaphore* is based on the principle of multiple pass through one semaphore. To facilitate multiple pass, we have to secure verification of the rights to pass. This verification also has to be atomic. From this requirement the number of necessary semaphores is obvious – we need two ordinary semaphores. The Mutex class, which was described in (Alsonso *et al.*, 2007), serves as the model semaphore implementation. One semaphore is responsible for atomicity of the operation of verification at the entry to and the departure from the critical section. We call this semaphore slave. The main semaphore, which controls the access to shared resources, is called master.

For verification we introduced variable *owner_id* to the promiscuous semaphore. This variable identifies the owner of the semaphore. The owner of the semaphore is checked at every attempt to pass through the promiscuous semaphore. *HW_MUTEX_NO_PROCESS_ID* signals that the semaphore has no owner process.

We proposed a simple outline of the aspect for the promiscuous semaphore *(Code 1)*. It contains four methods – *create*(), *dispose*(), *acquire*(), *release*(). Methods *create*() and *dispose*() are used for creation and destruction of semaphores, and initialization of auxiliary variables. They are bound to the *binding_object_class*() pointcut, through which they enrich object constructor and destructor of the class, to which the aspect is bound. The aspect is bound to the class or classes that are stated in the *binding_object_class*() pointcut. Notice that current solution creates new semaphore for every object, to which it is bound, therefore, if only one semaphore for a group of objects is desired, it is necessary to use the key word static with the aspect variables. To create control semaphores only once, at the initialization of the first object, to which they are bound, the Singleton design pattern is used and whole aspect is marked as static.

For the aspect it is necessary to specify the *acquire*() and *release*() methods, which define the behaviour of promiscuous semaphore at the entry to and the departure from the critical section. The *acquire*() method secures permeability of the semaphore for multiple pass if the owner of semaphore is confirmed (promiscuous mode), inserts processes into the process queue, assigns the owner to the semaphore, and counts the passes through the semaphore *(Code 2)*. First, it checks if the identification number of the applicant matches the identification number of the semaphore owner. If the numbers are the same, it is the case of multiple pass through the semaphore and the pass counter is raised (bypass). If the numbers are different, the process is registered in the waiting list of the master semaphore. The master semaphore controls the access to the shared resource and only allows one process at the time to be registered as the owner. After the process passes the master semaphore, it is registered as the owner, the pass counter is raised and the semaphore owner is set.

```
void acquire() {
  int currect_id = Thread.get_current_id();

  //mutex is owned by same thread
  slave->acquire();
  if (owner_id == currect_id) {
    counter++;
    slave->release();
    return; }

  //mutex is free or owned by other thread
  else {
   slave->release();
   master->acquire();
   slave->acquire();
   owner_id = currect_id;
   counter++;
   slave->release();}
}
```

Code 2 Operation for critical section entry acquisition.

The *release()* method counts the number of departures from the critical section, and in the case of final departure, it removes the owner of the semaphore and allows next applicant to pass (*Code 3*).

```
void release() {
  int currect_id = Thread.get_current_id();
  slave->acquire();
  counter--;

  //set no owner on last release()
  if (counter == 0) {
    owner_id = HW_MUTEX_NO_PROCESS_ID;
    master->release();}

  slave->release();
}
```

Code 3 Operation for departure from critical section.

The *acquire()* and *release()* methods were designed so as to fulfil conditions of mutual exclusion to the maximum possible level. The conditions of mutual exclusion are partly handled by the *Mutex* class, the implementation of which is considerably dependent on the resources provided by operating system. Protected sections of code are defined through *protected_functions()* virtual pointcut override.

2.2 Control of access to shared resources

We chose communication line for illustration because communication line operations directly access the shared resources. Since we have well designed separation of concerns in the program, and there are no concerns tangled or scattered in the class *Communication_Line*, we can say that every operation in this class needs to be protected and only can be used by one process at the time. Therefore, we applied the

abstract aspect (*Code 1*, *Code 2*, *Code 3*) to concrete pointcuts to create a concrete aspect for communication line protection (*Code 4*).

```
aspect Promiscuous_Communication_Mutex
  : public Promiscuous_Mutex {

  //virtual pointcut override
  pointcut binding_object_class()
    = "Communication_Line";
  pointcut protected_functions()
    = "% Communication_Line::%(...)"
      && ! static;
}
```

Code 4 Derived aspect for communication line protection.

It is possible to use means of protection implemented via aspects implementing the semaphore (*Code 1*) for control of access to shared resources. The aspects are implemented as abstract with overrideable pointcuts, which enables us to apply them to arbitrary concern of embedded system. The aspect implementing the promiscuous semaphore (*Code 1*) is robust, and with well designed pointcuts, no need for change of the definitions of concrete pointcuts arises.

3 MAIN RESULTS

We used existing software of the device, which supported several types of operations and transactions. This software was also deployed in the live operation, therefore, it provides proper testing material. We only modified the software in the field of semaphores and shared resources access control. For a better conception of how important the shared resources access control is, here is the number of operations executed in one transaction (*Online payment with PIN*). On average, in one transaction there are:

1. 4 – 10 communication line uses,
2. 20 – 50 NVRAM operations,
3. 3 – 5 Mailbox communication processes,
4. Multiple access to a printer,
5. Multiple access to a keyboard,
6. Multiple access to a screen

An example of multiple access to the keyboard is the case when the PIN entry dialog is running. This dialog uses the crypto-processor of the device, the functions of which require a dedicated process and exclusive access to the keyboard.

We executed the testing with four configurations. In *the first configuration*, we removed all shared resources access control mechanisms from the program. In *the second configuration*, we used plain code of the application, which serves as the reference point for other configurations. For *the third configuration*, we created six concrete aspects derived from the abstract aspect for the simple semaphore (Alsonso *et al.*, 2007). For *the fourth configuration*, we created six concrete aspects derived from the abstract aspect for the promiscuous semaphore (*Code 1*).

We tested the *fault tolerance* by executing 50 attempts at transaction "Online payment with PIN" and monitoring execution success rate of these transactions. Transactions were considered successfully executed if they got all the way from the beginning to the end and the device returned to a stable state afterwards. Moreover, we monitored parameters *average processor usage* and *program size after compilation*. *Operating memory usage* during testing could not be monitored because development environment did not offer suitable tool for this task. Test results are summarized in the table (*Table 1)*.

Configuration	Processor usage (%)	Failed transactions (max. 50)	Program size (kB)
No semaphores	34,72	40	2547
Tangled semaphores	27,22	0	2630
Simple semaphore	27,35	15	2555
Promiscuous semaphore	27,54	0	2615

Table 1 Measured values for individual configurations.

Based on the given results, we concluded that promiscuous semaphore in the form of aspect is viable. Processor requirements increased only slightly while the maximum security and fault tolerance were retained, which is a satisfying result.

4 CONCLUSION

We proposed our own solution for mutual exclusion (*Code 1*) that allows deadlock avoidance even when multiple call by the same process occurs. We applied this approach to real POS terminal application (*Section 3*). Based on acquired results of performance comparison of different approaches (*Table 1*), we declared our approach to be the best for given application.

One of the advantages of our approach is low pointcut fragility. We tested the robustness of concrete aspect (*Code 4*) weaved into the communication line by adding functions for internal modem operation. The concrete aspect was weaved correctly and worked according to the expectations, which confirmed the robustness of the solution.

Promiscuous semaphore aspect (*Code 1*) does not operate over the context of the join point (it operates over the context of caller process), and its advices are applicable to arbitrary pointcut. The critical section control advice is applied symmetrically, i.e., the lock and unlock functions are applied in pair, and with our approach there is no risk of deadlock due to forgotten unlock or lock call.

We utilized aspects also to affect memory protection. Through concretization and weaving of our aspect (*Code 1*) for memory access protection, we achieved zero read/write fault rate. This experiment extended the ways of memory protection using aspects (Hoffer *et al*., 2007), where the possibility of protection via weaved semaphore was not mentioned before.

REFERENCES

Afonso, F., Montenegro, S., Silva, C., Tavares, A.: Applying aspects to a real-time embedded operating system. In: *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. ACM, (2007).

Afonso, F., Brito, N., Montenegro, S., Silva, C., Tavares, A.: Aspect-oriented fault tolerance for real-time embedded systems. In: *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*. ACM, (2008).

Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A.: Efficient embedded software design with synchronous models. In: *Proceedings of the 5th ACM international conference on Embedded software*. ACM, (2005), pp. 187 – 190.

Deng, G., Gokhale, A., Schmidt, D. C.: Addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems via aspect-oriented & model-driven software development. In: *Proceedings of the 28th international conference on Software engineering*. ACM, (2006), pp. 811 – 814.

Hofer, W., Lohmann, D., Schröder-Preikschat, W., Spinczyk, O., Streicher, J.: Configurable memory protection by aspects. In: *Proceedings of the 4th workshop on Programming languages and operating systems*. ACM, (2007), article no. 3.

Kopetz, H.: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, (1997).