ESCORT: Lessons from an Integration Project

Andrea Savigni †‡ Filippo Cunsolo ♦ Francesco Tisato †‡

† Consorzio Milano Ricerche
Milan, Italy

‡ DISCO - Università di Milano-Bicocca
Milan, Italy
{savigni|tisato}@disco.unimib.it

♦ Project Automation S.p.A.
Monza, Italy
filippo.cunsolo@p-a.it

October 26, 2000

1. Background and Goals of the Project

ESCORT (European Standard Controller with Advanced Road Traffic Sensors) Project TR 4008 is a multinational collaborative project within the European Commission Research and Development Framework Programme IV, DG XII Telematics Applications Transport Research. The project began in January 1998 and ended in March 2000.
ESCORT deals with the development of a new philosophy of traffic control, based on *integration of heterogeneous devices and applications* for traffic control at intersection level. Thus, contrary to most projects in the traffic control area, ESCORT is not concerned with improving traffic by defining new algorithms, devices or applications. Rather, its main focuses are on *software engineering* and particularly *integration*.
This paper describes the AMI (Abstract Model of Intersection) that constitutes the core result of the project. The official ESCORT Web pages can be found at:
`http://www.soton.ac.uk/~trgwww/escort/index.htm`

1.1. The "Pre-ESCORT" Situation in Intersection Control

Traffic control at the intersection level is taken care of by a number of devices and applications. Among the devices the most notable ones are:

- traffic lights;
- local traffic controller(s), devoted to controlling traffic lights. There is usually one traffic controller per intersection, but very large and complicated intersections may have two (or even more);
- inductive loops, located below the surface of the road, whose job is to detect the presence of vehicles over them;
- cameras, set on poles overlooking the intersection. They usually measure two main quantities:
  - intersection occupancy i.e., the percentage of the intersection occupied by vehicles;
  - queue length i.e., the number of vehicles waiting in line before a traffic light.

The applications taken into consideration in ESCORT are:

- VBC (Video-Based Control): an AI application that, based on the queue length measured by the cameras, takes real-time control decisions on the local controller (e.g., by directly changing colours or adjusting timings of the current phase). VBC is the only application concerned with control;
- AID (Automatic Incident Detection): based on images taken by cameras and current state of lights determines whether an accident is present. The notion of accident considered by AID is actually quite broad in that it not only considers collisions among vehicles but, more generally, it treats as incident any vehicle standing still in “forbidden” locations, such as the centre of an intersection;
- VES (Vehicle Enforcement System): detects vehicles going through the red light.

The pre-ESCORT situation in intersection control, summarised in Figure 1, is characterised by *verticality*.

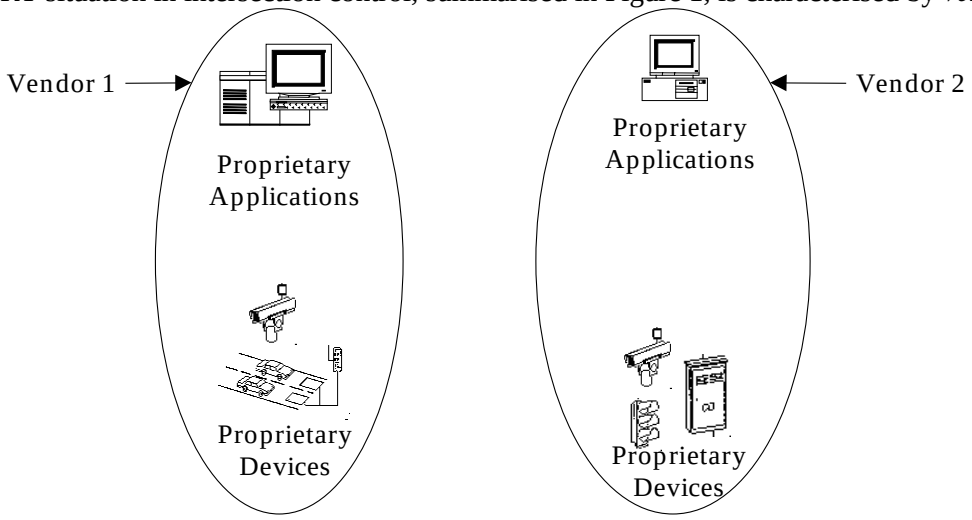

Figure 1. The pre-ESCORT situation.

Proprietary applications are monolithic and intermix domain-oriented and device-dependent issues. This means that a few vendors are free to monopolise the market by selling complete, vertical, proprietary solutions, from devices up to applications. Once one is committed to a particular vendor, there is no chance to switch to other technologies other than throwing away all past investments.

1.2. The Overall Goal of ESCORT

The initial goal of the project was to have the three applications described above run on devices from different vendors. After some debate, a general approach was chosen i.e., to establish an integration platform, called AMI (Abstract Model of Intersection) and described in some detail later, to be used as a reference platform for new devices and applications. Once (or maybe *if*) the AMI becomes an accepted industrial standard, application developers and device vendors will be able to build "AMI-compliant" products. This will allow integration among heterogeneous, multi-vendor devices and applications.

Since in the short term the project is concerned with integrating *existing* devices and applications, two more layers proved necessary:

- IWA (Interface with Applications), between AMI and applications, made up by *application managers*, each of which interfaces AMI with an existing application. Therefore, applications can behave as if they directly interfaced the corresponding proprietary devices; in other words, applications should not be changed in any way for them to work with AMI;
- IWD (Interface with Devices), between AMI and devices, containing *device managers*, each of which interfaces AMI with an existing device. As for applications, devices should not be aware of the existence of the AMI, and should behave exactly as they did without it.

The resulting product (AMI+IWA+IWD) is called SIM/IPS (Standard Interfacing Module/Intersection Platform Software), and is the overall outcome of the project. The SIM/IPS is a working product and is being tested in three actual intersections in Milan, Paris, and Valencia. The "post-ESCORT" situation is shown in Figure 2.

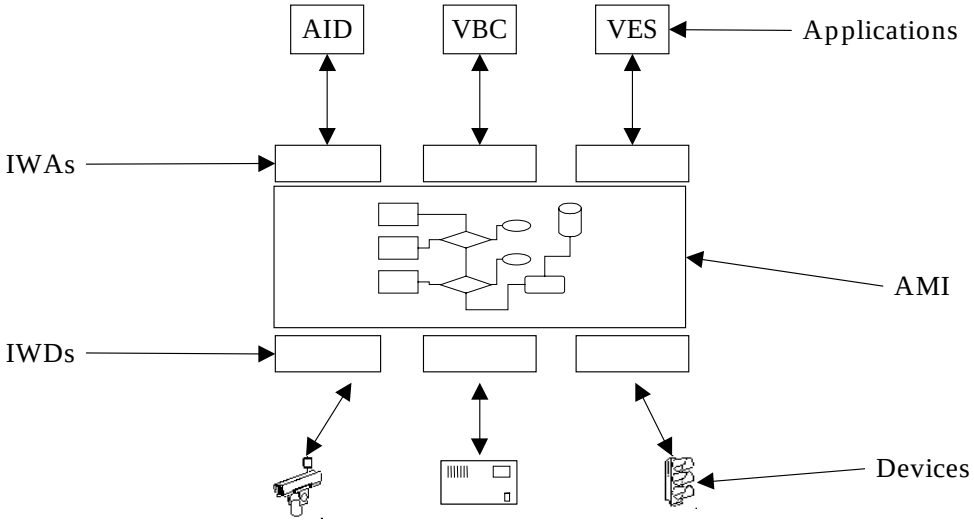

Figure 2. The "Post-ESCORT" situation.

1.3. The ESCORT Consortium

The ESCORT Consortium is composed by both industrial and academic partners, coming from five different European countries (Belgium, France, Italy, Spain, UK). As we will also emphasise below, one of the key difficulties encountered within the project was the diversity of partners' background. Without getting into the details of each partner's role, we will briefly summarise here the main areas of expertise that were involved.

- Traffic control. Partners skilled in this area were coming from companies, research institutions, and academic departments. People working in this area tend to be very clever at designing algorithms and techniques for optimising traffic control strategies. Some of them are also really good at designing image processing algorithms; for example, some sell cameras equipped with on-board cards that can measure queue lengths, intersection occupancy, or that can even track vehicles in order to build O/D (Origin/Destination) matrices. However, they usually have no or little idea as to architect a software system in order to achieve separation of concerns, reuse, and other desirable properties. In other words, they lack a *software engineering* culture.
- Software engineering. The partner skilled in this area was coming from an academic department. Its role was to devise a system architecture that was flexible enough to accommodate present and future devices and applications. Needless to say, the software engineering partner had no idea of traffic control problems.
- "Bare metal programming." The industrial partners employ implementers who are used in programming very low-level devices. Even though they are very good at that, they lack software engineering good practices, and tend to build highly-efficient, but monolithic and *ad hoc* systems.

These differences led to daily clashes among the different partners, particularly between software engineering people, who were constantly heading towards reuse and clean design, and bare-metal programmers, who insisted on efficiency and "quick-and-dirty" coding.

2. The AMI Design

2.1. Requirements and Overall Approach

The AMI is the heart of the ESCORT project. The AMI is centred around an Abstract Model of the Intersection, which constitutes the integration platform through which applications and devices communicate. This model was conceived to meet several different requirements:

- completeness. Since very diverse applications and devices are expected to rely on it, the AMI must represent, at an appropriate abstraction level, all the elements that can be found in the underlying physical intersection;

- extensibility. Even if the project itself required integrating three applications and a well determined number of devices, the AMI in theory should accommodate new devices and/or applications without being changed. However, should the need arise for a new piece of functionality, this should be integrated seamlessly into it without changing the existing parts;
- efficiency. Some devices and applications (e.g., the VBC) need to access information and/or issue commands within tight time limits.

An object-oriented approach was employed throughout the project lifecycle, and the UML was chosen as the modelling and design language. Space limitations and industrial secret prevent us from getting into the details of the model. Thus, only some excerpts of the most important diagrams will be shown. Finally, always for the same reasons, in class diagrams attributes and methods were intentionally omitted.

2.2. The Packages

The AMI is made up by four packages: `Static`, `Logical`, `Management`, and `Control`. Figure 3 shows the overall relationships between the AMI packages, on the one side, and the interfaces to devices and applications, on the other side. All of the dependency relationships are one-way; this means that the AMI is never aware of IWDs or IWAs. In other words, the AMI is a server for these, that in turn behave as clients (see section 4 for implementation details).

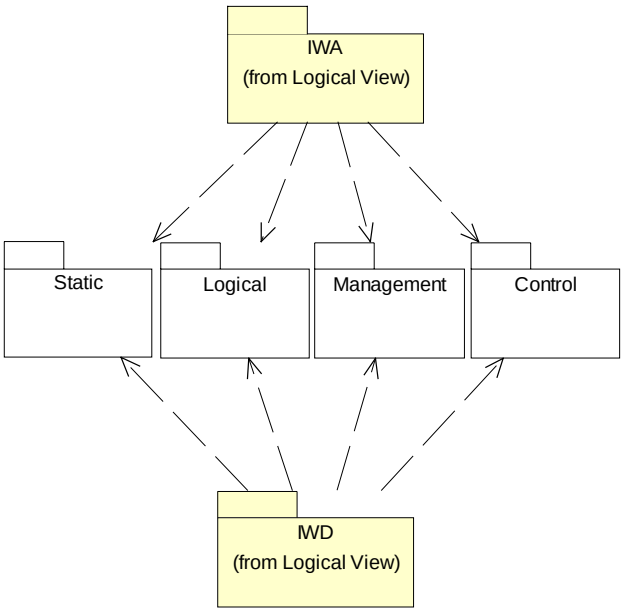

Figure 3. The AMI packages.

2.2.1. The `Static` package

Contains the topological model of the intersection. It represents concepts such as the intersection itself, lanes, etc. It is not meant to change during program execution, hence its name (see section 4 for a brief coverage of configuration issues). Figure 4 shows an excerpt of the package structure.

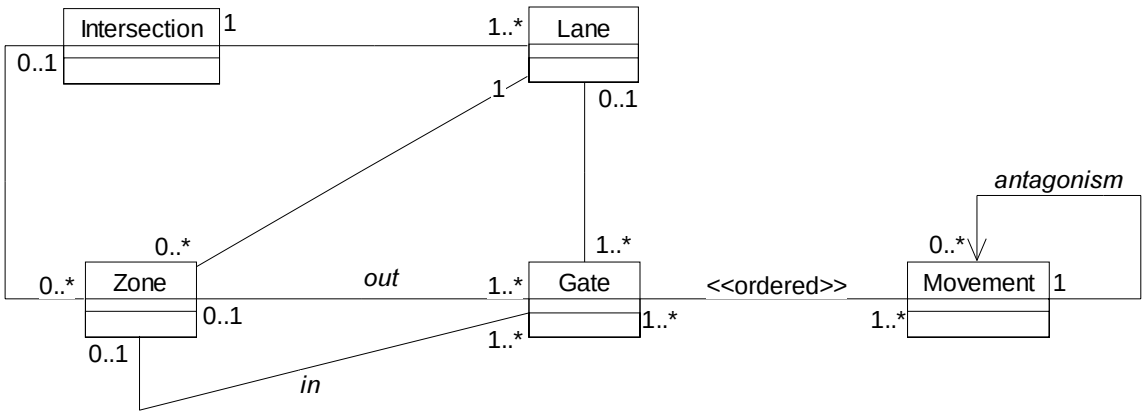

Figure 4. An excerpt of the `Static` package.

A few classes appearing in the diagram deserve a brief explanation:

- `Gate`: models a surface through which vehicles flow. Gates are set up off-line and don't change over time. They are abstract entities that can be implemented by any sort of physical device capable of measuring a flow (cameras, inductive loops, etc.) or can have no implementation at all, thus simply serving as zone delimiters;
- `Zone`: represents a rectangular area that can be monitored by a sensor. A zone is always delimited by at least one `in` gate and one `out` gate;
- `Movement`: models a *possible* movement, expressed as an ordered sequence of gates. Movements are very useful for example to define conflicting movements, which in turn are essential for setting up traffic plans.

2.2.2. The `Logical` package

Contains the representation of dynamic information where state can be observed (e.g., traffic flow through a gate) or controlled (e.g., traffic positions, that define which vehicle movements are allowed at a given time). These abstract concepts allow applications to observe and control the state of the intersection without any knowledge about the physical devices.

Consider, as an example, the following scenario. Traffic flow is most likely measured by either a camera or an inductive loop. However, in this package we are not interested in the devices, but rather in the logical representation of an observable flow through a logical gate. Thus, this package contains a class called `GateSensor` but *not* a `Camera` nor a `Loop` class (which are to be found in the `Management` package under the names of `FieldVideoSensor` and `FieldLoop` respectively). `GateSensor` is a logical representation of any device seen, at a high abstraction layer, as a flow measurer. This allows applications to interact with abstract concepts regardless of the underlying (and possibly changing) technology implementing them. Figure 5 shows a small but meaningful part of the `Logical` package that is related to these abstract sensors. Note the relationships existing with the classes imported from the `Static` package; in a sense, entities in the `Logical` package add an observable or controllable state to static objects.

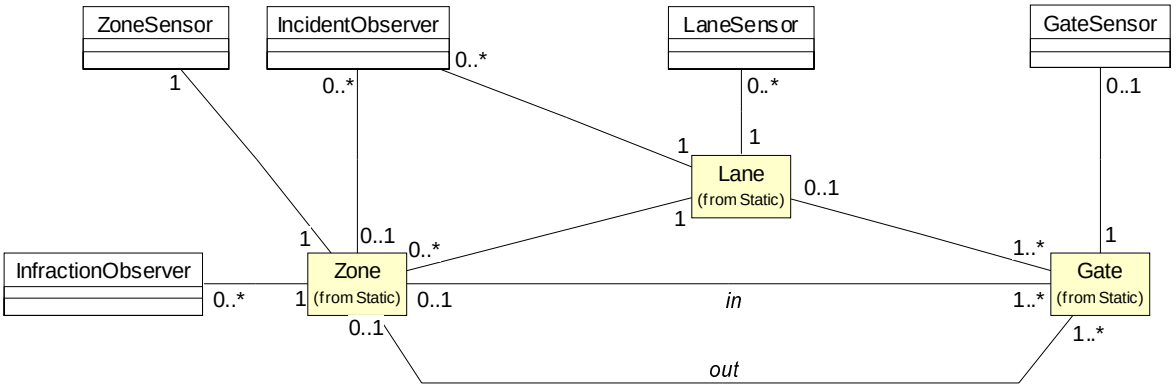

Figure 5. One excerpt of the `Logical` package.

The `Logical` package also contains classes that are more directly related to traffic control (see Figure 6).

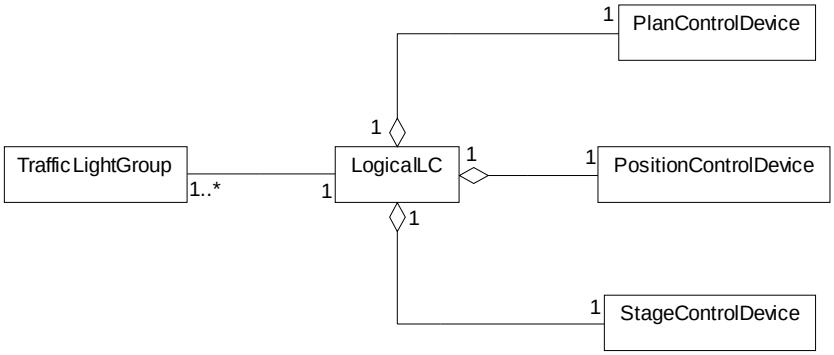

Figure 6. Another excerpt of the `Logical` package.

The central class here is `LogicalLC`, an abstraction of the traffic controller that resides in the intersection. A `LogicalLC` is modelled as an aggregation of a number of virtual devices devoted to controlling the various aspects of traffic, such as plans, stages, etc, that belong in the `Control` package. `LogicalLC` is also associated to the fundamental class called `TrafficLightGroup`, that represents a set of traffic lights that are bound to always have the same colour (in fact, in order to prevent possible discrepancies, the traffic lights that make up a TLG are typically connected to the controller by a single cable). This concept was modelled to take into account the fact that traffic controllers, for control purposes, do not reason in terms of single traffic lights but in terms of more general light groups.

2.2.3. The `Management` package

Figure 7 shows an excerpt of the Management package class diagram.

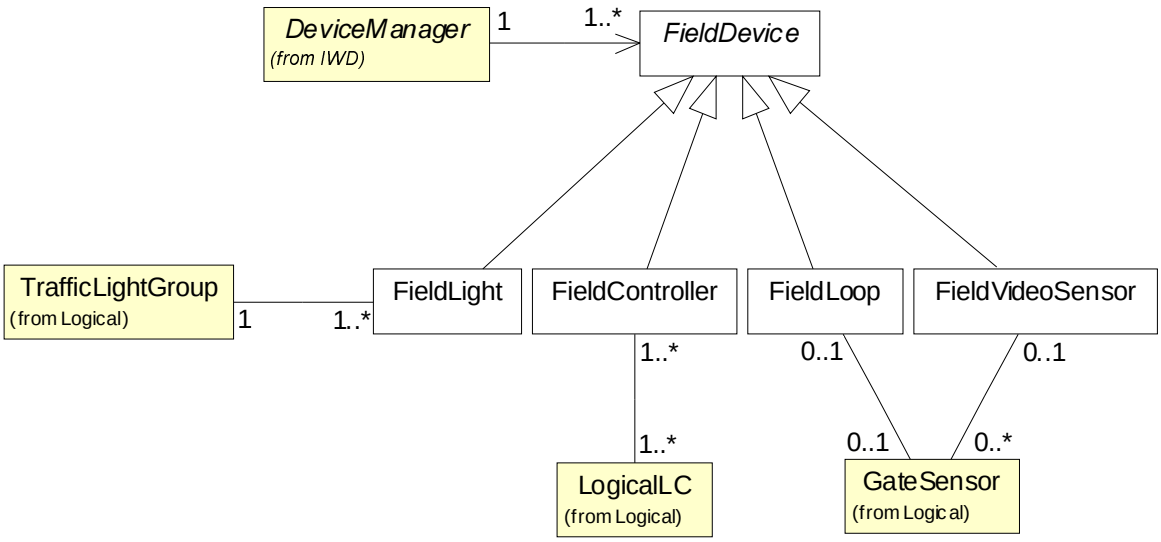

Figure 7. An excerpt of the `Management` package.

Classes in this package represent the physical counterparts of those in the `Logical` package. In particular:

- `FieldLight` models a traffic light (in the usual sense);
- `FieldController` provides a physical view of the local traffic controller(s) in the intersection, as opposed to `LogicalLC` (`Logical` package) that represents its logical function;
- `FieldLoop` represents an inductive loop;
- `FieldVideoSensor` models a camera. Note that a `GateSensor` (that represents, as explained above, the logical view of a flow measurer) can be associated to either a `FieldLoop` or a `FieldVideoSensor`.

Entities in the real world (cameras, loops, traffic lights, etc.) have thus two representations: a logical one, which emphasises the logical function of the entity (so for example a `GateSensor` is seen as something that is capable of measuring a flow) and a physical one, essential for management purposes. The physical view of a gate, for example, contains static identification data (make, model, etc.) along with state information that is directly related to the functioning of the device, such as working/non working, and to its tuning.

Keeping these views separate is essential for the following two reasons:

- separation of concerns. If an application needs to know the queue length in a given lane, it need not know (or rather, it *must* not know) about the actual technology employed. However, a management application and/or a human technician need to be able to tune applications and devices and thus need to know all the details about their internal workings;
- cardinality. The same real-world entity can be represented by *n* logical entities and by *m* physical ones. Recall the example of a `TLG` that represents a group of traffic lights that are bound to always have the same colour. Each of the traffic lights in a `TLG` is represented as a single entity in the `Management` package (`FieldLight` class) because, for instance, traffic light bulbs burn independently of one another.

2.2.4. The `Control` Package

Contains all the entities that are strictly related to traffic control, such as plans, stages, etc. Since these are very specialised concepts, they will not be further covered.

3. The Dynamic Behaviour

The system's dynamic behaviour was conceived with reuse in mind. This led to the adoption of the following principles:

- it must be possible for the designers to define the system's dynamic behaviour without changing the AMI static models (class diagrams, package structure, etc.). This is a basic requirement because each installation (present and future) must meet specific constraints due either to the applications or to the devices; for example, different local controllers can have *very* contrasting time requirements. Thus:
- all entities in the AMI are passive objects, i.e., they do not have an autonomous thread of execution. The result is that the whole AMI does not have a thread of control. All time-related functionality is performed outside the AMI. In particular, the local controller clock was chosen as the reference clock for the whole devices-AMI-applications chain. Both the AMI and the applications synchronise on that;
- each Manager, be it a device manager or an application manager, has an autonomous thread of execution;
- a significant state change of an AMI entity generates an event. The `Observer` pattern is employed for notifications from the AMI. Device and application managers can register on AMI entities and be notified when something relevant happens. Figure 8 shows a (very) simple example of asynchronous notification: application managers can register on devices and thus be notified when their state changes. Note that, as a general rule, a "pure event" approach was chosen i.e., the notification carries little or no state information; it is up to the application to request AMI entities for more data. This choice was made in order to make asynchronous notifications as fast as possible. However, in cases such as the one depicted in Figure 9, in which the state information associated with the event is very small, this can be embedded in the notification.

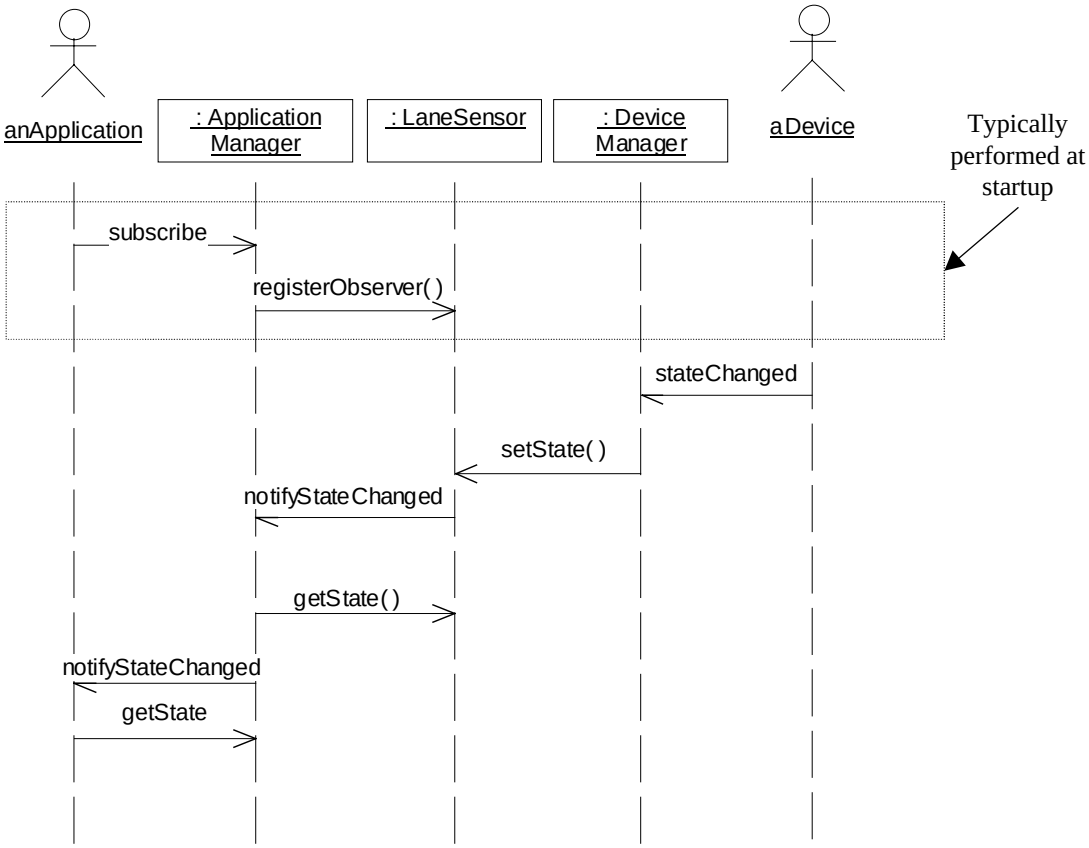

Figure 8. A very simple example of asynchronous notification.

As far as real-time issues are concerned, the partners did not deem it necessary to employ specific real-time technology, such as real-time kernels and/or specialised hardware. The practical tests proved them right. The standard, commercial technology employed gave good results and performed within the time requirements.
Finally, one important issue is the distinction between *expected* and *actual* states. Consider the sequence diagram in Figure 9. The VBC application often needs to control the state of the local controller. However, for safety rea-

sons, the controller needs to reach a reference configuration before changing state (a typical such configuration is “all red lights”), which may take some time. This yields the need for a separation between expected state (the state the application wants the controller to be set to) and actual state (the state the controller is actually in), and for associated notifications.
This pattern was employed for a number of attributes of several devices.

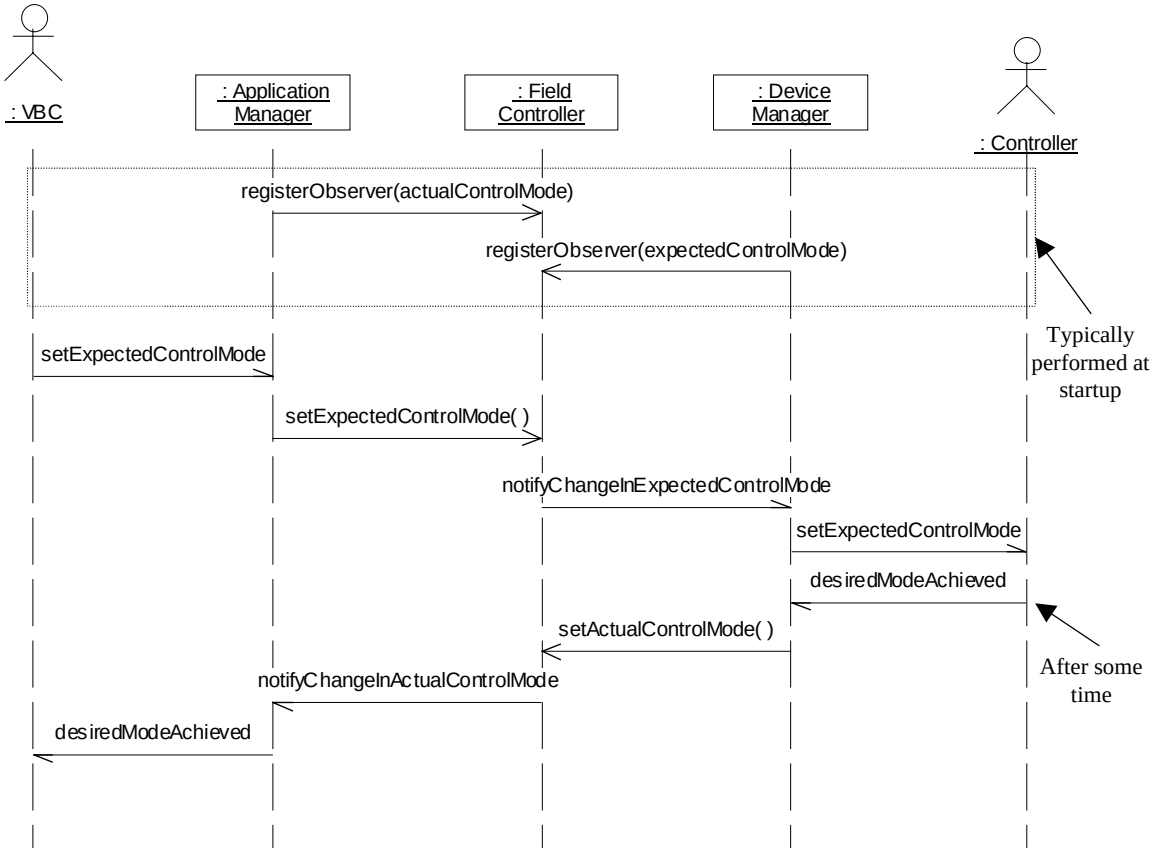

Figure 9. An example of *actual* vs. *expected* modes.

4. Implementation Issues

The partners agreed on a C++ implementation using Microsoft’s COM technology on Windows NT. A component-based technology was deemed most appropriate for this project due to the strict and clean separation between the AMI and its clients (potentially unknown in number and technology), that must not know about AMI internals.
The AMI was implemented as a COM component made up by one executable file (which is little more than a startup utility) and four DLLs (corresponding to the four packages). Clients (IWD and IWA) can follow a simple, standard procedure (extensively documented in the AMI user manual) to connect to the AMI and use its services. This implementation allows developers in the future to change a package without the others noticing, and even adding more packages in a seamless way.
In order to make the interactions between AMI and clients as clean and simple as possible, the concept of *container* was introduced. A container is a COM interface that provides access to all the objects of a certain class. So for example there exist a `ZoneSensorContainer`, a `TLGContainer`, and so on. Choosing the granularity of access to AMI entities was a controversial point. In the end, a compromise solution was adopted. Containers were created for all packages but for some of them, namely `Management` and `Logical`, some partners insisted that another access mode be added. So for these two packages it is also possible to access entities indi-

vidually; in other words, in these two packages all entities are COM components, even if it is always possible to manipulate them via containers.
Another grouping mechanism is *clusters*. A cluster is a group of homogeneous entities on which operations can be performed simultaneously and, most important, atomically. Clusters were introduced because some applications, namely the VBC, need to update the state of a group of devices without being interrupted.
Finally, since the AMI was not conceived to be a non-stopping application, dynamic reconfiguration was not implemented. Therefore, configuration is performed as an off-line, static activity. A configuration data base was designed that is read at startup by the AMI. Application and device managers, in turn, retrieve configuration data directly from the AMI. This has two desirable effects:

- all configuration activities are centralised, which avoids data duplication;
- application and device managers need not access the configuration DB, and are thus completely unaware of the actual DB technology employed. All startup information is obtained through the AMI.

5. Conclusions and lessons learned

ESCORT was a complex project, for a number of reasons:

- diversity of partners' background. As already explained above, difficulties arose between partners with a strong object technology background and others strongly focused on optimisation issues and highly experienced in "bare metal programming." Many of the key design decisions were the result of compromises between the two parties;
- integrating existing systems is notoriously harder than building them anew. While the long-term goal of ESCORT is to provide a common platform for new systems, its short-term goal was to integrate existing ones. Thus, a very hard modelling work was necessary in order to take into account the very diverse existing devices and applications;
- the initial scheduling was quite tight and actually proved *too* tight: a three-month extension was requested to, and obtained by, the EU. This delay seems frankly quite reasonable for a two-year project.

Despite the above difficulties, the project was successfully completed. The system is fully implemented and working in all of the three intersections. Two annual reviews were completed. The last one, performed in January 2000, was especially successful, and received very good ratings by the EU committee. The whole system in a simulated environment (comprising Valencia controller simulator, IWD to Valencia controller, AMI, IWA to the GUI, GUI, application simulators) was demonstrated on a small laptop (32 Mb RAM!).
Under many respects, ESCORT was a very instructive project, from which many lessons were learned:

- probably there was no need of proving that, but conceptual modelling turned out to be the real key issue in the overall project lifecycle. As stated above, integration was the major force in the project, and integrating requires, first of all, understanding and representing the existing. Under this respect, the UML proved (once again) precious even in this very technical, specialised field. This proved wrong the many people who maintain that the UML is only good for business-oriented software. In particular, the class diagram-sequence diagram pair (the details of which we cannot report here due to space limitations) proved particularly good for interface specification;
- the design activity was underestimated. While a lot of effort was (rightly) put in the analysis model, the same effort was not spent in the design phase. The result was that, also due to pressure put by some of the partners, the implementation phase started prematurely, based on a model that was not detailed enough. This made the implementation work fairly hard. In particular, since some important design decisions were not taken in the design phase, developers had to make a decision at coding time. This, combined with the physical distribution of the partners, led to some divergence among the different packages with the inevitable consequence of integration problems;
- object orientation is a very popular buzzword but a largely ignored practice. Everybody these days claims to be object-oriented but this project, among others, taught us that *actual* object-oriented practice is still out of reach of many software professionals. In particular, we found that the very foundation of object orientation, namely the separation between interface and implementation, is still a goal to be achieved. As a matter of the fact, one of the partners insisted on developing type libraries (i.e., interfaces in COM parlance) and code at

the same time. This led to a huge delay in the delivery of interfaces which of course impacted on all the other partners and on the project as a whole.