ESCORT: Lessons From an Integration Project

Andrea Savigni - Università di Milano-Bicocca. Milan, Italy
Filippo Cunsolo - Project Automation S.P.A. Monza, Italy
Francesco Tisato - Università di Milano-Bicocca. Milan, Italy

The ESCORT Project

- ESCORT: European standard controller with advanced road traffic sensors
  - Begun in January 1998 and ended in march 2000
  - Partners from Belgium, France, Italy, Spain, UK
- The goal: a new philosophy of traffic control at intersection level
  - Integration of heterogeneous devices and applications for traffic control at intersection level
  - Using object technology (UML+CBSD)

Traffic Control at Intersection Level: a Software Engineer's Perspective

- A number of devices...
  - Traffic lights
  - Local traffic controller(s)
  - Inductive loops
  - Cameras
- ...and applications...
  - Intelligent traffic control
  - Incident detection
  - Vehicle enforcement
- ...that don't talk to each other!

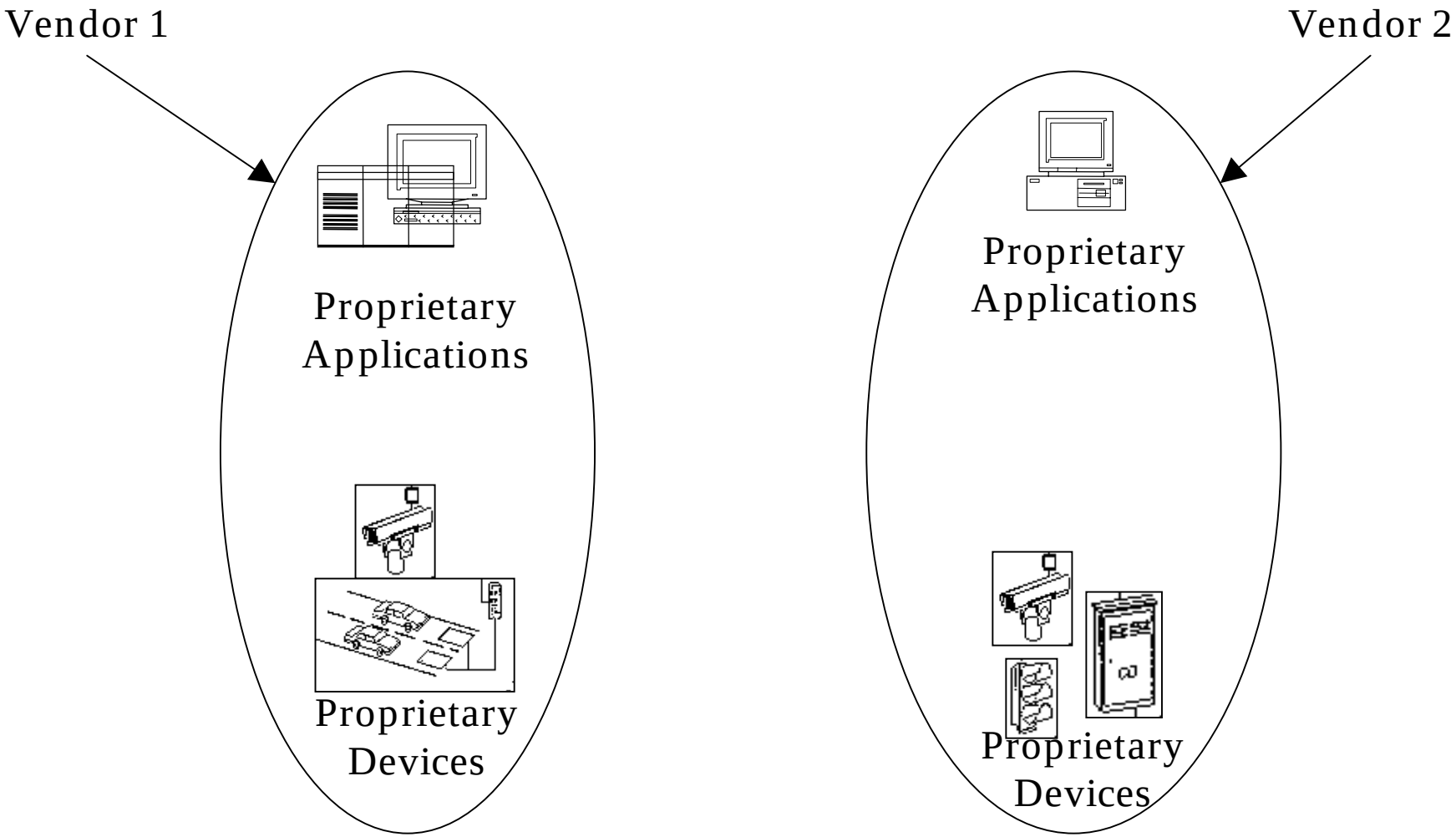
Traffic Control at Intersection Level: a Software Engineer's Perspective (Cont.)

- An archaic situation
  - No integration
  - Applications are vertical and monolithic
  - No separation of concerns between domain-oriented and device-dependent issues
- Market monopoly
  - A few vendors sell complete, vertical, proprietary solutions, from devices up to applications
  - No mixed-vendor combinations possible

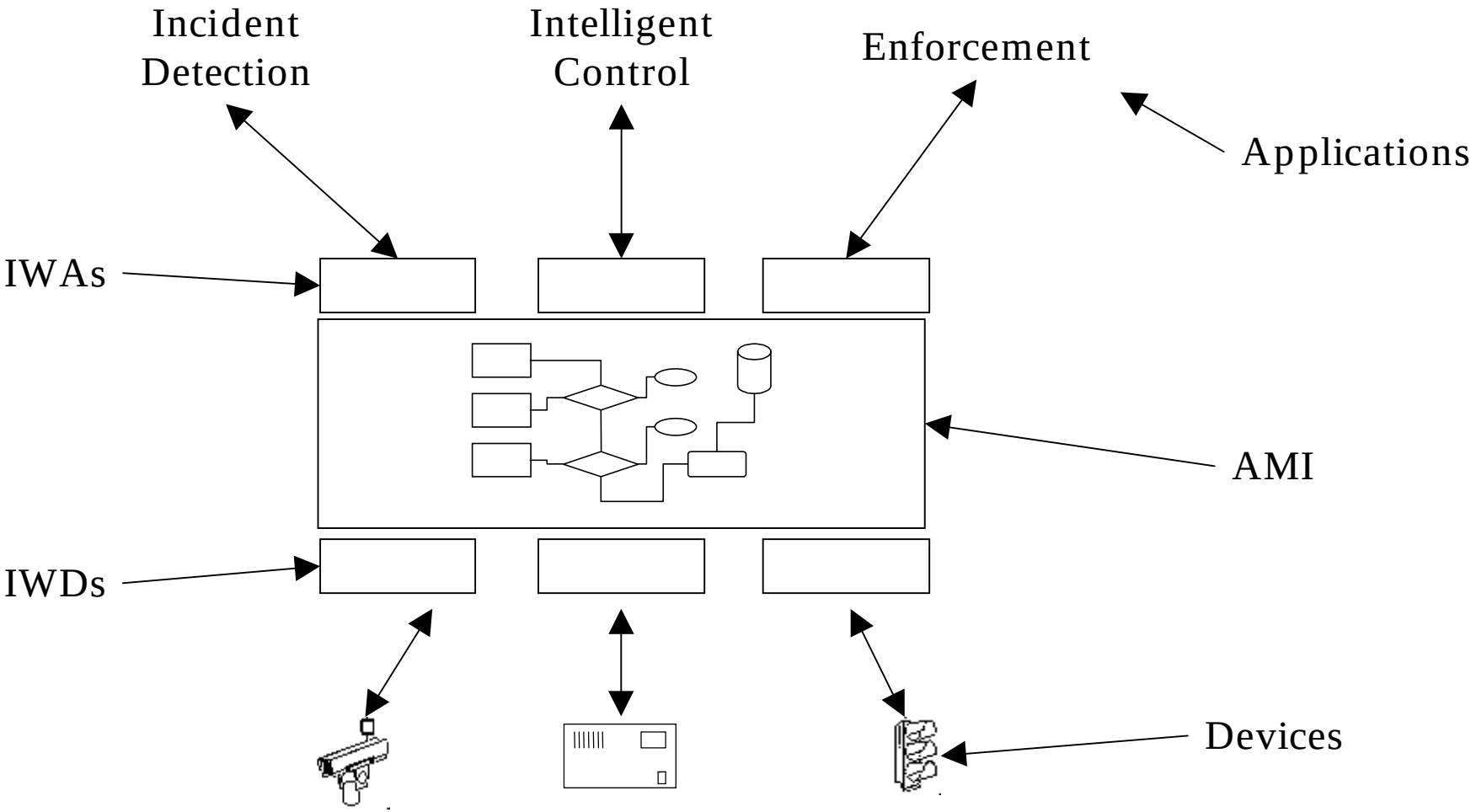
The Heart of the Project

- The AMI (Abstract Model of Intersection)
  - A reference platform for integrating diverse, multi-vendor, present and future devices and applications
- In the long term:
  - New devices and applications built against the AMI ("AMI-compliant" from scratch)
- In the short term:
  - Integrating *existing* devices and applications
  - Two more layers necessary: IWD (interface with devices) and IWA (interface with applications)

Before ESCORT



After ESCORT



The Forces

- Technological. The AMI must be:
  - Complete: must represent *all* the entities in the intersection
  - Extensible: ready for future devices and applications
  - Efficient: some devices and applications have real-time requirements
- Managerial
  - Diversity of partners' background
    - traffic experts
    - software engineers
    - "bare metal" programmers
  - Physical distribution (a *key* issue)

AMI: the Packages

- `Static`:
  - The topological model of the intersection
  - Contains classes such as `Lane` and `Zone` that are not meant to change during program execution

AMI: the Packages (Cont.)

- `Logical`:
  - Contains an *abstract* representation of observable or controllable quantities (e.g., traffic flow, intersection occupancy, allowed vehicle movements, ...)
  - Applications can observe and control the intersection without any knowledge about the underlying physical devices
  - Implementation can be changed (e.g., from loops to cameras) in a completely transparent way

AMI: the Packages (Cont.)

- `Management`:
  - Physical counterpart of the `Logical` package
  - Contains classes such as `Camera`, `Loop`, ...
  - Models state information related to the functioning and tuning of *concrete* devices

- `Control`:
  - Contains highly-specialised, traffic control-related concepts

Why Two Representations?

- Why a logical *and* a physical representation?
- Separation of concerns
  - applications must *not* know about abstract devices' implementation (which may thus be changed at any time)
- Cardinality
  - the same real-world entity may be represented by *n* logical entities and by *m* physical ones

The Dynamic Behaviour

- The AMI must be *one* and *general*:
  - the very same piece of software must run on each installation
  - but each installation has a different local controller
- The AMI dynamic behaviour must be independent of the static models

The Dynamic Behaviour (Cont.)

- The solution: delegate time-related issues to the local controller
  - The local controller holds the reference clock
- All AMI entities are *passive* objects
- Publish / Subscribe model

Implementation Issues

- C++/COM implementation on Windows NT

- AMI implementation:
  - One .exe file (basically a startup utility)
  - Four DLLs, corresponding to the four packages
  - Extensible: new packages can be added seamlessly

- Why a component-based technology?
  - Clean separation between AMI and clients, potentially unknown in number and technology
  - Language-independent (test programs developed in VB)

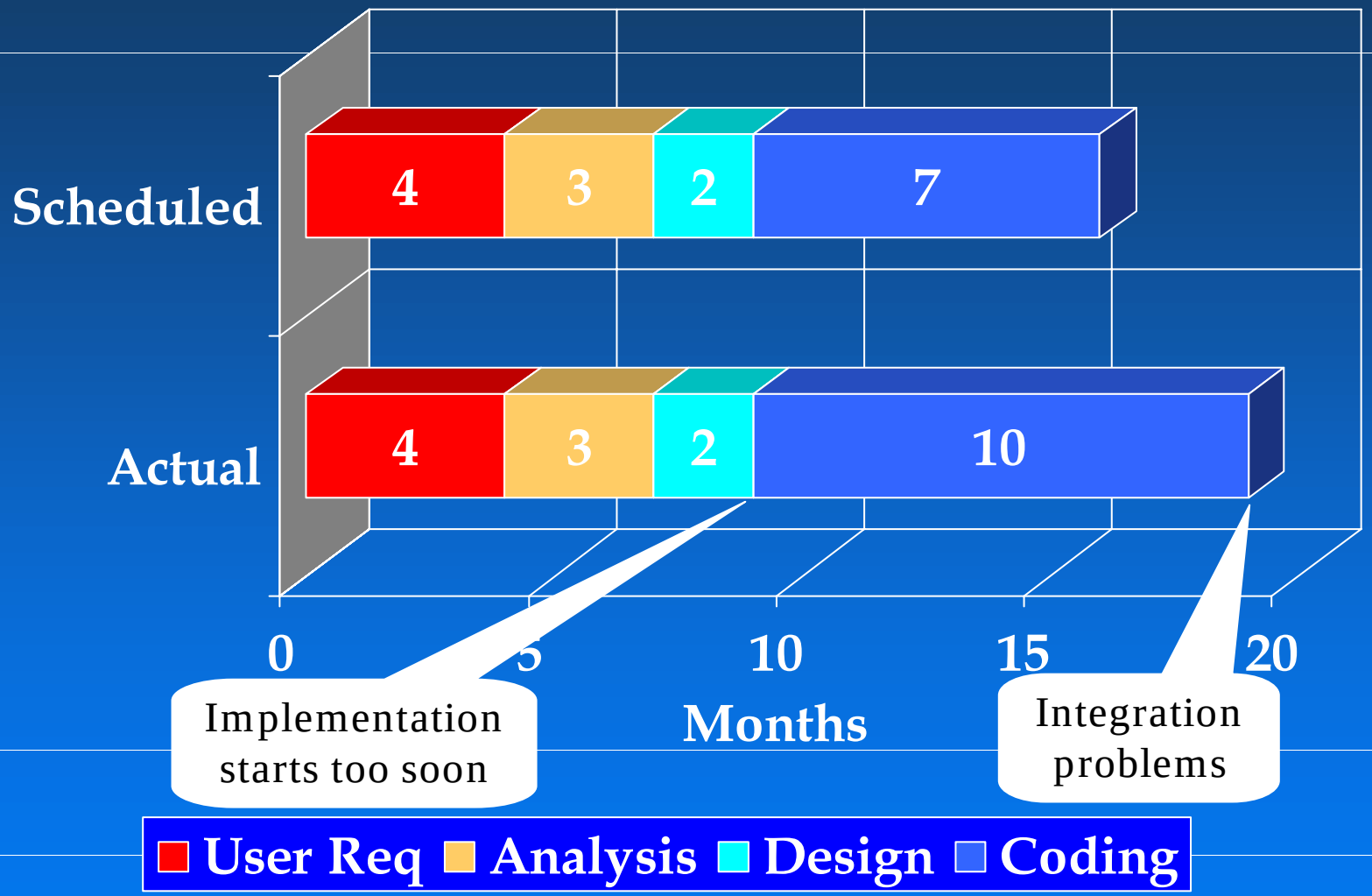
Problems Encountered (Technical)

- Diversity of partners' background
  - daily clashes between "reuse people" and "optimisation people"
  - most key design decisions resulted from compromises between the two parties
- Integrating is harder than building anew
  - ESCORT's short-tem goal: integrating *existing* devices and applications
  - *very* hard analysis work to model the very diverse existing devices and applications

Problems Encountered (Managerial)

- Initial scheduling too tight
- Three-month extension obtained by the EU
- Physical distribution
  - "one meeting is worth one thousand mail messages"

The Timing


Scheduled
4
3
2
7
Actual
4
3
2
10
0
5
10
15
20
Months
Implementation starts too soon
Integration problems
User Req
Analysis
Design
Coding

Results Achieved

- AMI + IWA + IWD is working right now in three European intersections
  - Milan, Italy - Paris, France - Valencia, Spain
  - Each with its own equipment and devices
- Two EU annual reviews passed with very good ratings

International Events

- OOPSLA - for the software engineering approach
- EWGT (Euro Working Group on Transport) - for traffic control-specific solutions
- IST2000-Nice - one of the four EU projects selected by the European Commission for exhibition and demonstration

Lessons Learned

- Conceptual modelling is the really key issue in the overall project lifecycle
  - the good analysis job made the complexity of the domain manageable
- The UML works well even in very technical, specialised fields
  - not only in business-oriented software

Lessons Learned (Cont.)

- Design is too often allocated insufficient resources
  - very careful analysis, but too cursory design
  - many key design decisions taken at implementation time
  - integration problems

Conclusions

- The greatest single lesson learned from ESCORT is:

Resist managers and bare-metal programmers

Invest time in building a conceptual model before even thinking of implementing