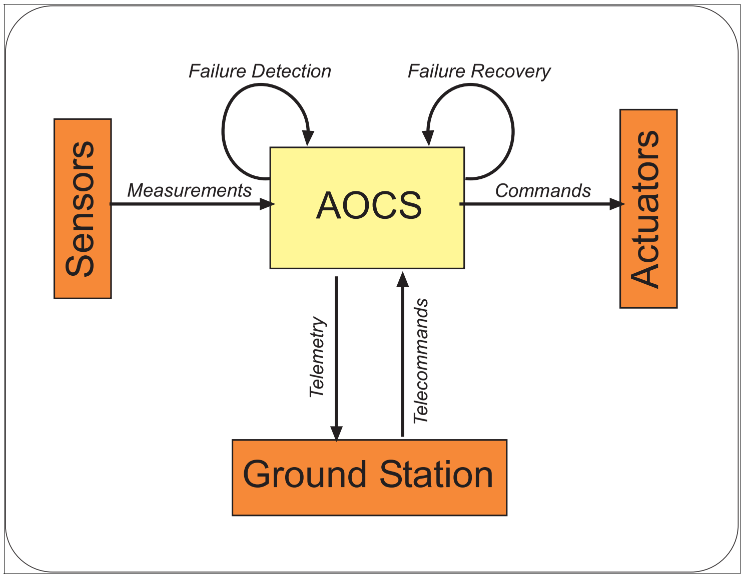
The AOCS Framework Project

Wolfgang Pree, Alessandro Pasetti, Tim Brown
Department of Computer Science
University of Constance
{*FirstName.LastName*@uni-konstanz.de}

Contents

- The AOCS (Attitude and Orbit Control System) (3 slides)
- Software Frameworks (4 slides)
- The AOCS Framework (16 slides)
- Current & Future Activities (2 slide)

AOCS Structure


Failure Detection
Failure Recovery
Sensors
Measurements
AOCS
Commands
Actuators
Telemetry
Telecommands
Ground Station

NB: The AOCS is much more than just implementation of control algorithms!

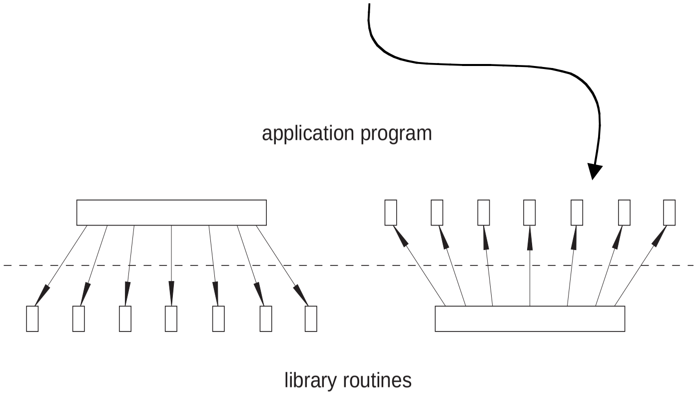
Background to AOCS Project

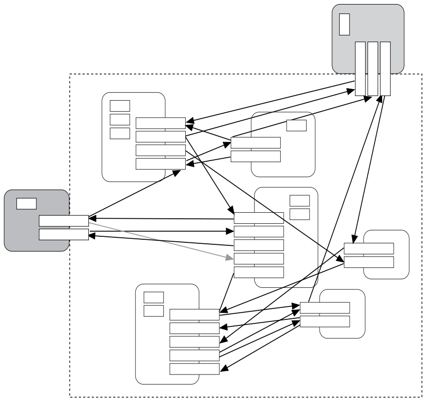
- Current Situation:
  - AOCS software is typically “made to order”
  - AOCS sw (like most embedded sw) is technologically not state-of-the-art: C/Ada83 + modular paradigm
- In Aug. 1999 ESA placed a contract with SRL to investigate sw reuse for the AOCS
  - Adoption of framework technology
  - Design and development of a prototype framework for the AOCS completed in Dec. 2000
  - Design team included both sw and AOCS (control systems) expertise

Definition of the term *framework*

Framework :=

A piece of software that is extensible through the callback style of programming





Pros and cons of frameworks

Frameworks...

- \+ allow **reuse of architecture design + code**
  - => significantly reduced development and maintenance costs
  - => standardized application structure
- \+ can be produced for almost any commercial and technical application domain
- – require a significant development effort (requires detailed domain knowledge)
  - => long-term investment
  - => pay-off if similar applications are developed for a domain
- – are at odds with current project culture (one project, one application)

Definition of the term *component*

Component :=

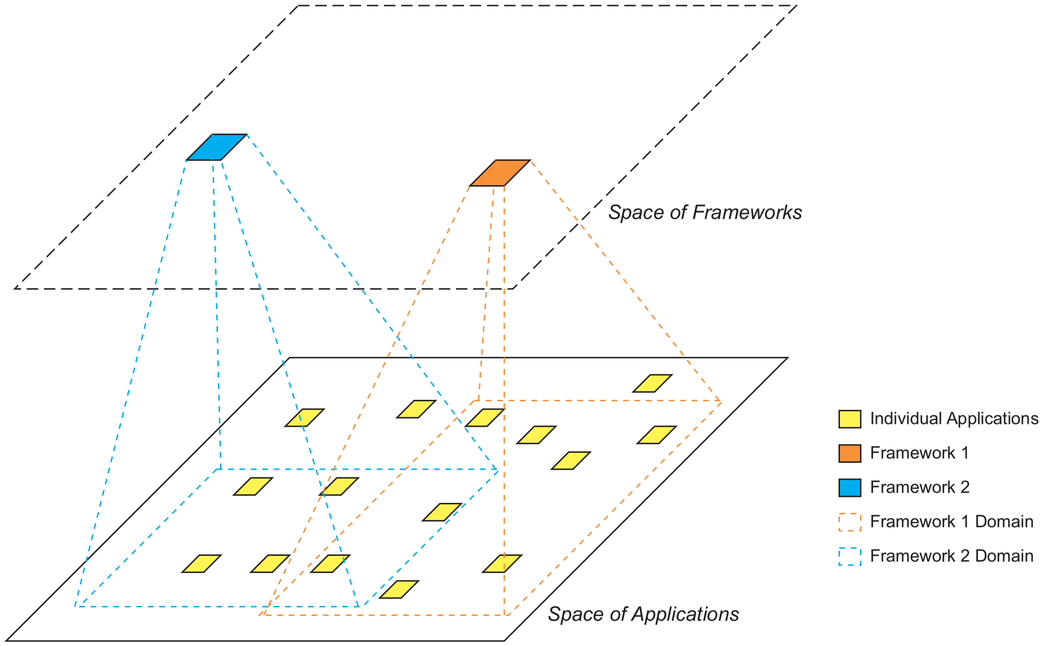
A piece of software with a programming interface

Consequences:

- frameworks that offer a programming interface are components
- module-oriented languages (Modula, Oberon, Ada) and component standards (CORBA, COM, JavaBeans) just offer different ways of defining such programming interfaces

---

Another View of Frameworks

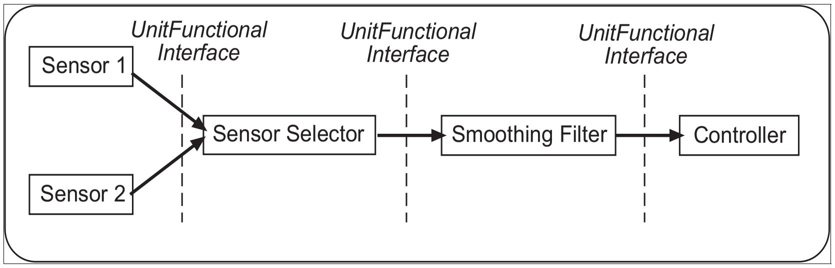


Some AOCS Design Aspects - 1

- The AOCS is made up of independent components that cooperate by exchanging data …
  - Shared memory modelled on Linda/JavaSpace
- AOCS components need to monitor each other to detect failures and to synchronize their behaviour …
  - Three monitoring mechanisms modelled on JavaBeans property monitoring: direct monitoring, conditional monitoring, monitoring with notification
- AOCS components exhibit mode-dependent behaviour …
  - Modified version of the “Strategy Pattern” from Gamma *et al*

Some AOCS Design Aspects - 2

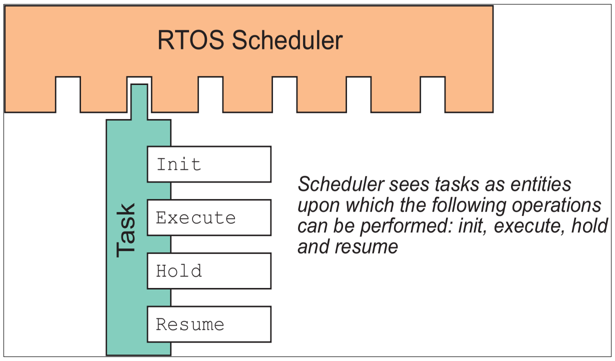
- AOCS need to implement data processing chains …
  - Block/Superblock mechanism mimicking the similarly named concepts in MatrixX/Xmath
- Data from sensors and to actuators need to go through several processing stages …
  - Definition of `UnitFunctional` abstract interface to be implemented by each unit processing component



Reuse Approach : The RTOS Model

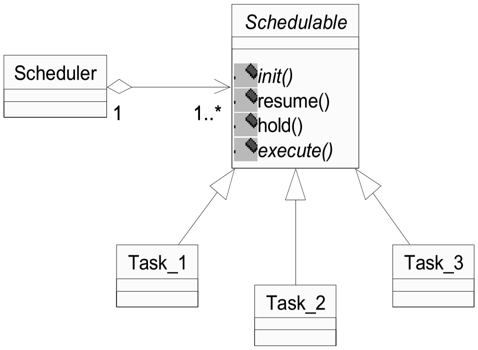
RTOS‘s are examples of reuse in real-time field → inspiration for AOCS f/w

Plug-In View



Scheduler sees tasks as entities upon which the following operations can be performed: init, execute, hold and resume

Class View



Task *management* separated from task *implementation* through an abstract i/f

---

The RTOS Example and the AOCS

- The RTOS example shows that the management of some functionalities – like task scheduling – can be packaged in reusable components

- For typical AOCS functionalities like:

  telecommand management, telemetry management, closed-loop controller management, failure detection management, failure recovery management, sensor\actuator management, etc

  Can application-independent (and hence reusable) functionality managers be constructed?

Reuse Approach for the AOCS

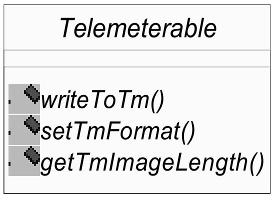
- Divide the AOCS into functionalities: TM management, TC management, unit management, FD management, FR management, etc.

- For each functionality:
  - Define an abstract interface separating the functionality management from its implementation
  - Build a reusable functionality manager component (application-independent component)
  - Build reusable components providing default implementations of recurring functionality implementations

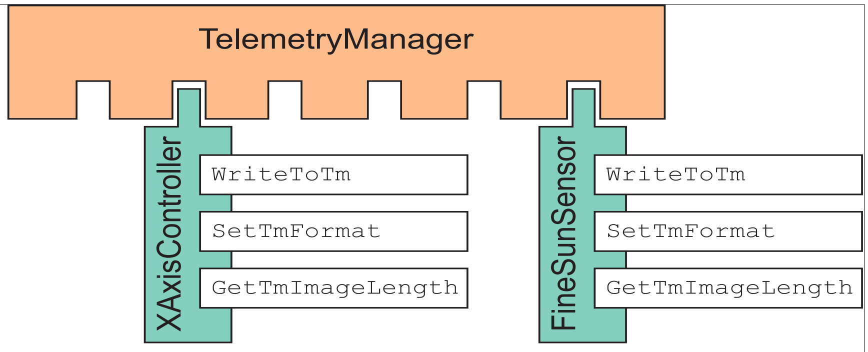
---

Telemetry Management Example

- Identify abstract operations required to handle telemetry:
  - `writeToTm()` : object writes its own state to the TM stream
  - `setTmFormat(newFmt)` : set the TM format to `newFmt`
  - `getTmImageLength()` : return the length (in bytes) of the object's TM image
- Define an abstract interface for telemetry operations:



Telemetry Manager

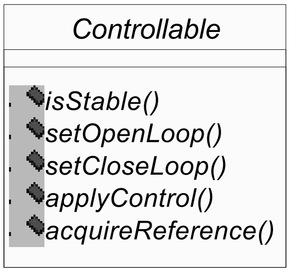


TM Manager maintains a list of pointers to objects of type Telemeterable and periodically processes them using the Telemeterable operations:

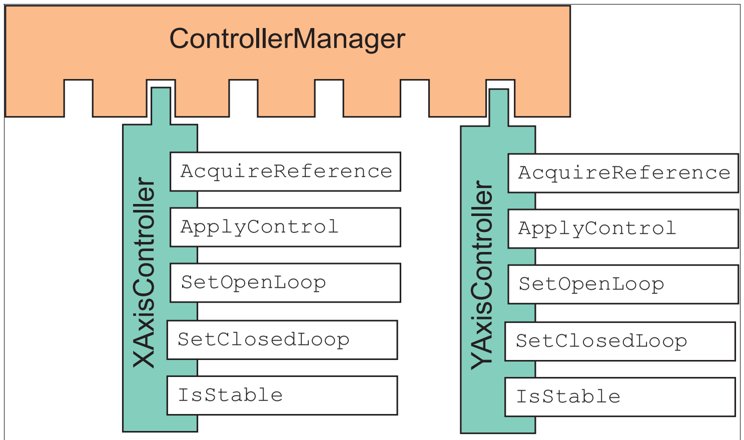
```
Telemeterable* list[N];
. . .
for ( i=0; i<N; i++ )
{   list[i]->setTmFormat(fmt);
    size = list[i]->getTmImageLength();
    if (image fits into TM buffer)
        list[i]->writeToTelemetry();
    else
        . . . // error!
}
```

Controller Management Example

- Identify abstract operations required to handle closed-loop controllers:
  - `acquireReference()` : acquire controller set-point
  - `applyControl()` : compute control torque and send to actuators
  - `setOpenLoop()/setClosedLoop()` : operate in open/closed control loop
  - `isStable()` : ask controller to check its own stability
- Define an abstract interface for controller operations:



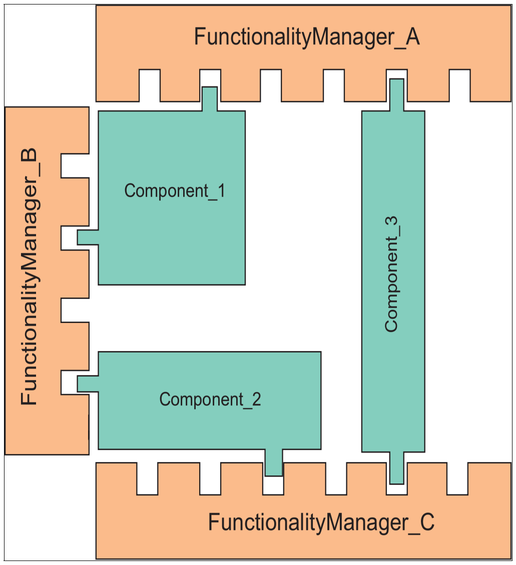
Closed-Loop Controller Manager



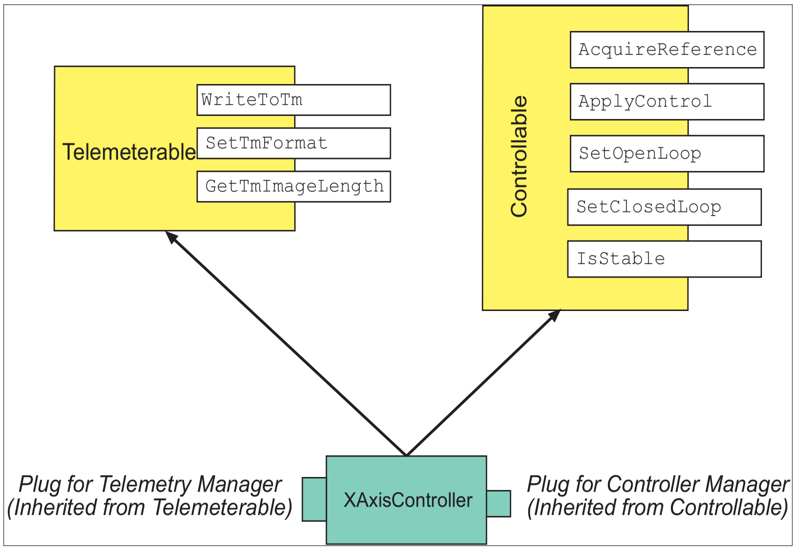
```
Controllable* list[N];
. . .
for ( i=0; i<N; i++ ) do
   if ( list[i]->isStable() )
   {   list[i]->acquireReference();
       list[i]->applyControl();
   }
   else
     . . . // error situation!
```

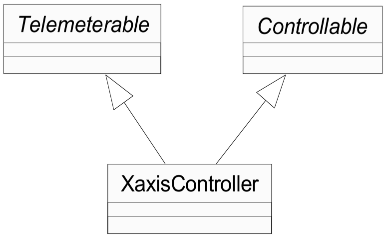
Conceptual AOCS Architecture

- Red blocks are application-independent and reusable
- Green blocks tailor generic architecture to needs of a specific application
- Each FM defines an abstract interface
- A component may contribute to tailoring several FM's



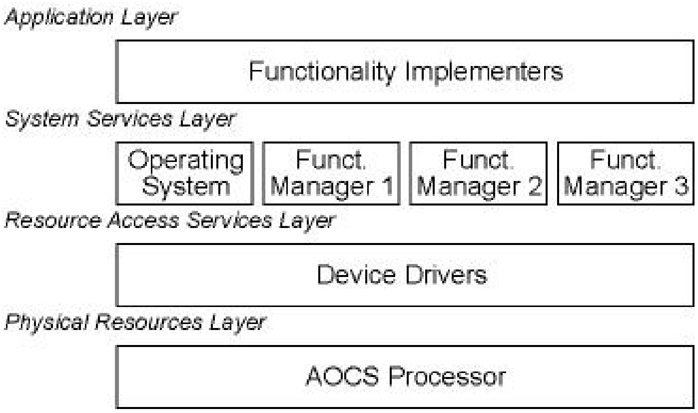
Multiple Interface Implementation





NB: Java-model of multiple inheritance is used (safe!)

FW as Domain-Specific OS Extension


Application Layer

Functionality Implementers

System Services Layer

Operating System | Funct. Manager 1 | Funct. Manager 2 | Funct. Manager 3

Resource Access Services Layer

Device Drivers

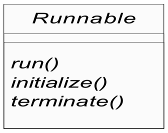
Physical Resources Layer

AOCS Processor

The (application-invariant) functionality managers can be seen as domain-specific extensions of the operating system

Scheduling Aspects

- The AOCS is assumed to be cyclical and is seen as a bundle of functionalities with each functionality manager implementing:



- Method `run` is called from outside the fw and it causes the actions associated to the current cycle to be executed.
- Functionality managers make no assumptions about how often or in what order they are called, such assumptions must be built into the (application-dependent) plug-in components
- The current version of the framework provides no default protection mechanisms for access to shared data

---

Summary of Reuse Model

- Domain-specific *design patterns* provide standard solutions to recurring design problems
- *Abstract interfaces* decouple functionality management from functionality implementation
- *Core components* encapsulate reusable functionality managers

  NB: functionality managers do not perform actions *upon* objects, rather they ask objects to perform actions *upon themselves*

- AOCS framework suitable in general for embedded control systems

Prototype Implementation

- Prototype implementation language: C++ (GNU compiler)
  - Any OO language can be used
  - Ada95 was considered but discarded due to poor support for MI
- No dynamic memory allocation
  - Non-trivial objects are created at initialization and never destroyed (no dangling pointers)
- No exceptions, no run-time type identification, …
  - Error situations are handled through creation of event objects in shared memory areas
- Target processor: ERC32 + RTEMS operating system
  - SPARC processor qualified for use in space by ESA (megabytes memory, ~ 10 MIPS@14 MHz)

Resource Requirements

- Timing requirements for “empty” functionality managers: 0.2 ms @ 14 MHz per AOCS cycle
  - This is the overhead introduced by the framework infrastructure
  - Typical AOCS cycle durations are 50-500 ms
- Memory requirements for functionality managers: 43 kB (code) + 19 kB (data)

- AOCS Prototype requirements (inclusive of RTEMS and C++ run time systems but with some modules missing):
  - 1 AOCS cycle in 3.9 ms
  - 245 kB (code) + 92 kB (data)
  - AOCS prototype not really representative of “real” AOCS

Current & Future Activities

- Contract with Nokia:
  - Enhancement of the AOCS framework (Novato) and application to other embedded control systems (eg, helicopter control system)
- Contracts under negotiations with ESA:
  - use the AOCS framework to develop and test AOCS for the Proba satellite (Proba is a mini-satellite to be launched in 2001 as a technology demonstration mission)
  - Port the AOCS framework to a real-time version of Java
    - Java is a “natural” implementation medium for sw frameworks
    - In the long-term, Java may be interesting as a language of choice for mission-critical software
- Contract proposal to Nokia:
  - Develop sw robustness techniques for OO systems (domain-specific compiler extensions)

Concluding Remarks

- F/Ls and ICs were successfully tested in the AOCS project
  - Use of F/Ls made design more manageable and will make it easier to extend the FW to other application domains (some F/Ls can be carried over unchanged to other domains)
  - ICs were the main source of design changes in the AOCS project
- F/Ls and ICs are being used as the core of a complete methodology for FW development

Potential Cooperation Objectives

- Apply AOCS framework to helicopter control software
  - not all functionality managers need to be implemented
  - structure of AOCS and helicopter control system seem similar
  - interface C++/Oberon?
- Use COM technology to interface framework components and to interface C++/Oberon components
- Implement framework-based helicopter control system software on Giotto infrastructure
  - Feasibility of integrating Giotto and AOCS has not been proven yet but is being studied