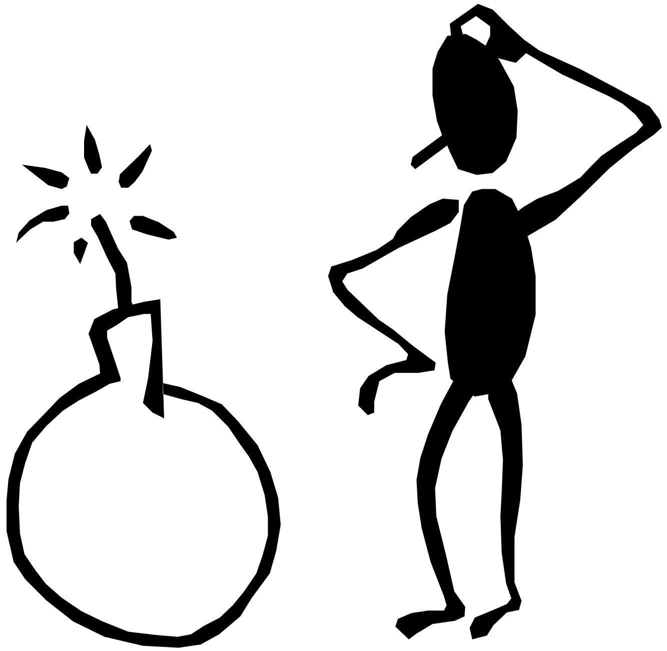
# Execution Architecture of a Real-Time system

Dr. A.P. Kostelijk

Ing. B. Smets



## Speaker Notes

Real Time systems are characterized such that *besides functional requirements also timing requirements should be met*. These timing requirements are usually *verified very late* in the development lifecycle and so result in a lot of rework in a late stadium.

In this presentation I will present the *different steps* to create an execution architecture for a real-time system, such that this *rework is reduced*.

## Contents

## Speaker Notes

When designing an execution architecture, one can think of *many resources* which are important and should be analysed in the run-time environment: e.g. memory usage, on screen display or graphics, etc. This presentation will focus on the processor (CPU) as a resource. So I will describe *techniques which can be used to analyse the run-time behaviour of the system*.

First the goals are explained to design a good execution architecture. Then three techniques are explained to reach the goal:

- a parametrizable model is shown
- a technique to use this model during the design steps.
- a technique which guarantees that timing requirements are met.

Then I will present the basic steps in the real-time analysis before concluding this presentation with the experience in the G+4 Set Top Box project.

# Intro: Context = Soni’s views

- Conceptual architecture
- Scenario’s (addition to Soni)
- Functional decomposition
- ***Execution architecture***
- Code architecture

## Intro: Goals (1)

- Related requirements for the execution architecture design:
  - Timing related:
    - **Meet timing requirements**
    - Prevent deadlocks
    - Prevent starvation

# Speaker Notes

During the architecture start-up phase, *Soni* (an expert in architectures from Siemens) shows that several architecture views are necessary to describe the architecture of your system.

The execution architecture view  describes the *dynamic structure of the system*. To create a good execution architecture it must be designed such that it is:

- flexible -> easy extendable (which is important for a platform), easy adaptable (different products derived from a platform have their own requirements, so also timing requirements)
- orthogonal to the functional architecture view. The functional architecture describes the functional decomposition and layering of the system.
- and last but not least that it guarantees that all timing requirements are met

To reach this goal, *three techniques* are described. Let’s start with the first technique : *create a parametrizable execution architecture using ‘g4events’*.

## Intro: Goals (2)

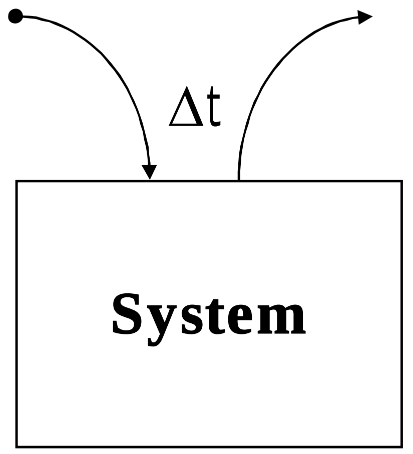
- Requirements for the execution architecture design:
  - others:
    - Decoupled from other views, to prevent rework multiplication.
    - Allow designers work in parallel.
    - Easily modifyable, because at first no performance times are known.

## Intro: Goals (3)

- Starting point:
    - orthogonal to functional architecture
    - flexibility: extendable and adaptable
    - guarantee timing requirements

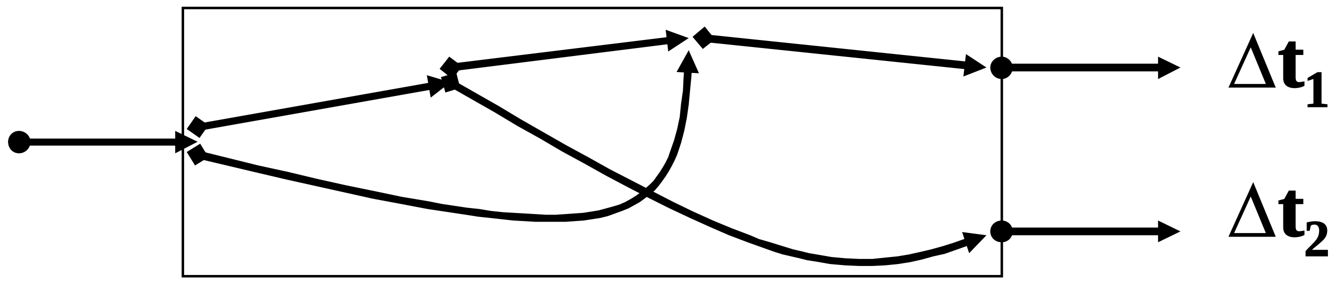
## What are timing requirements?

Event
Required deadline
Actual response



# What are timing requirements?

- Event:
  - external: signal: e.g. device or timer
  - internal: handover some datastructure
  - active or passive = interrupt or polling
  - Tree of events = action flow



## What are timing requirements?

A taxonomy of events:

- Cyclic
- A-cyclic
- Hard and soft response times
- Worst, best and typical case
- Very complex

## What are timing requirements?

- Concurrency is needed?
  - Idle processor can be utilised
  - Timing requirements must be met for all possible interleavings
  - RT-OS (pSOS, VX-WORKS, etc…)

## What is a (real-time) operating system?

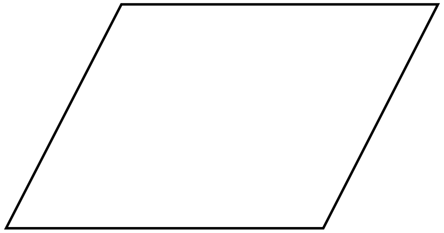
- Multi-tasking
- Timesharing versus multi-processing
- Cooperative versus preemptive scheduling
- Priority (RTOS) versus fairness (e.g. UNIX) based scheduling
- Task = process = thread, for today

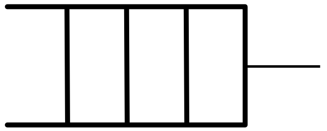
## What are timing requirements: what is the key?

- Focus on *events and action flow*, not on tasks.
- Task mapping is a derived implementation detail, neither a requirement nor a starting point.

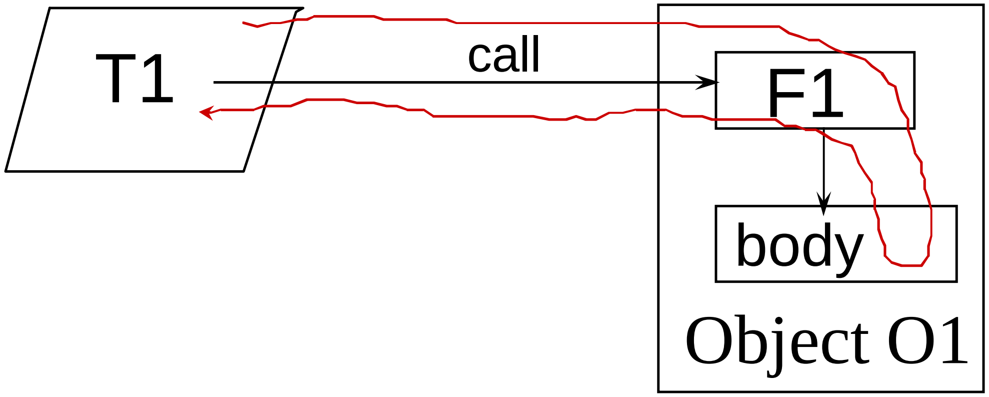
# Notation

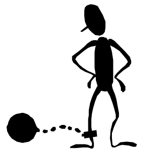
Encapsulation


Thread


Queue

# The ‘g4event’ model (1/6)





Objects / modules communicate via (member) functions

Synchronous execution

# Speaker Notes

A task T1 needs to execute somefunction and has a certain timing requirement. So when T1 is running it should respond within a certain time. Function F1 has an interface part ‘F1’ and a main body where the real work is done ‘body’. T1 executes F1 from object O1, which directly calls its body. Suppose now  that in the body of F1, we have to *wait for a hardware response*. This means that T1 is hold up for this hardware wait time and so it could be that the response time required for T1 is missed.

If we use this mechanism (objects communicate via functions), which aims at single threaded execution (as presented by the red line), we will be *clogged* when developing software. So we need to introduce concurrency in executing functions. As presented in the literature, for real-time systems also the complexity of the system decreases when multi-threading is introduced. (refers to main loop and a lot of dependencies).

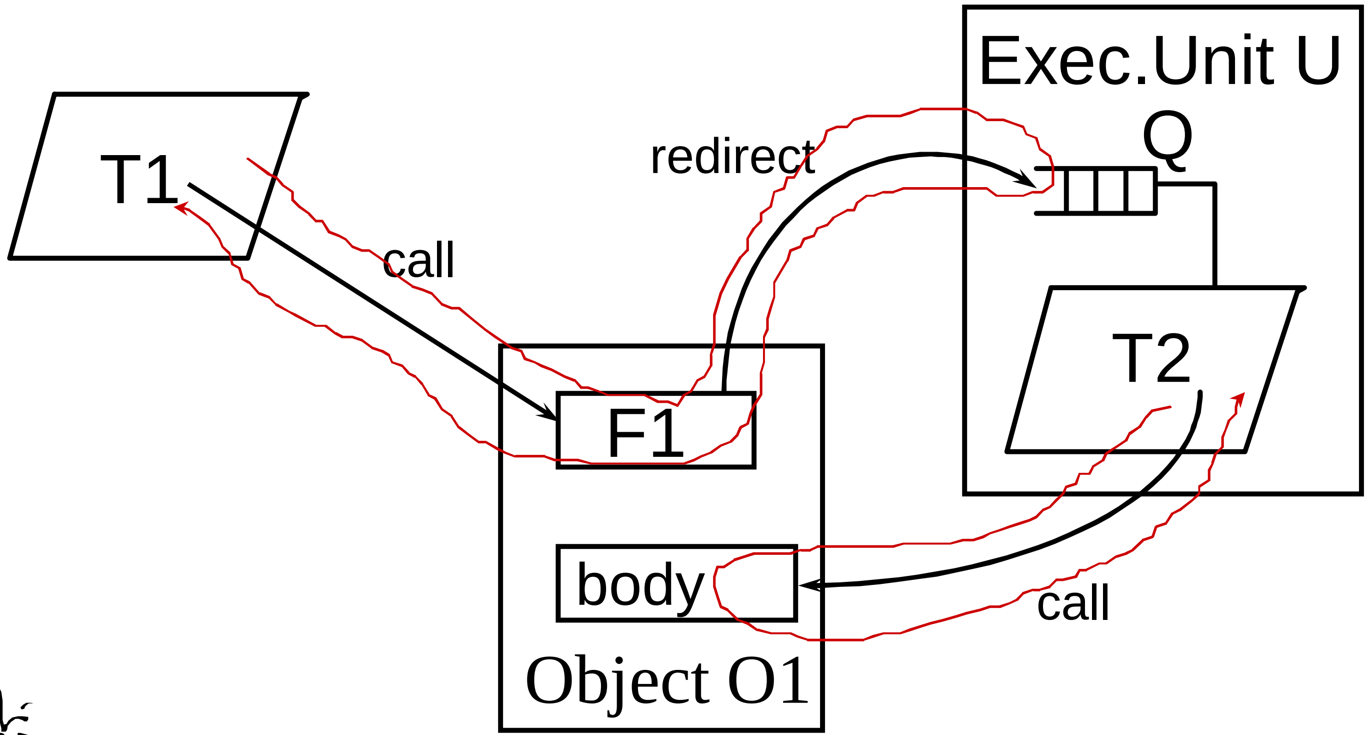
This mechanism can, however, be extended to multi-threaded execution easily as presented on the next slide.

(introducing concurrency give some more complications: mutual exclusion problem, task synchronization problem, producer/consumer problem)

## The ‘g4event’ model (2/6)

- Synchronous: The action of calling F1 returns after F1’s body is executed. (e.g. access to a diskdrive).
  - Object O1’s call and body run on Thread T1.
- Need concurrency to be flexible, and prevent idle time.

# The ‘g4event’ model (3/6)



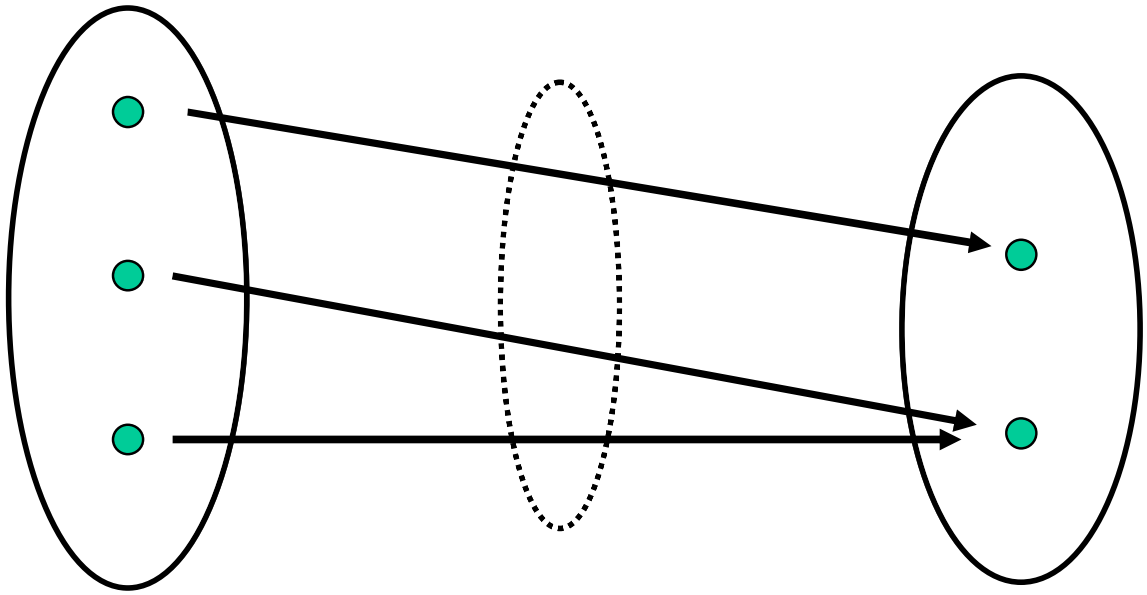
Active objects borrow a task from an Exec. Unit
Asynchronous execution

# Speaker Notes

We *introduce the notion of execution units* in the system. An execution unit is an object which encapsulates both a queue and a task. We have the same situation as explained in the previous slide. So task T1 executes function F1 of object O1. Now since Object O1 needs concurrency, the object is named an active object. An active object has a reference to an execution unit U. The interface part of F1 redirects the call to an execution unit. The execution unit puts the call in its Q and returns. T2 is scheduled and gets the message (call) which is in the Q. It then calls the body of Object O1. Note here that this body function is now  executed on T2 and not on T1 anymore. When we have to wait now for the hardware response, T2 is hold on for a while, but T1 is just continuing its execution and so can reach its timing requirements.

So using this *multi-threaded mechanism makes us happy* when designing a system to meet the timing requirements. We can say that active objects don’t have a task of their own, but they just borrow  one from an execution unit. By this means we can *design a parametrized architecture which is orthogonal on the functional architecture* view.

# The ‘g4event’ model (4/6)



Objects O

Exec.
Params P

Exec.
Units U

# Speaker Notes

With this ‘g4event’ mechanism, we can create a parametrizable execution architecture. All active objects contain a *reference (also named execution parameter)* to an execution unit.

If this reference is null, we have the single threaded call. We can also see that several active objects can be mapped to the same execution unit or that an active object can change from one execution unit to another execution unit  while the system is operating,  just by adapting the execution parameters.

Now  when we have specified such a parametrizable execution architecture, the question is how  can we use this.

## The ‘g4event’ model (5/6)

- all active objects (Oi) contain a reference (Pi) to an execution unit (Uj)
  - if reference is NULL -> single threaded
  - several Oi can be mapped to same Uj
  - an Oi can be mapped to several Uj during run-time
  - reference = execution parameter
- **execution** architecture is parametrized

## The ‘g4event’ model (6/6)

- Other issues for an event mechanism:
  - point-to-point versus broadcasting
  - push versus pull (subscription)
  - synchronous versus asynchronous
  - timer events (after, at, repeating, …)
  - send(e), receive(e), process(e)

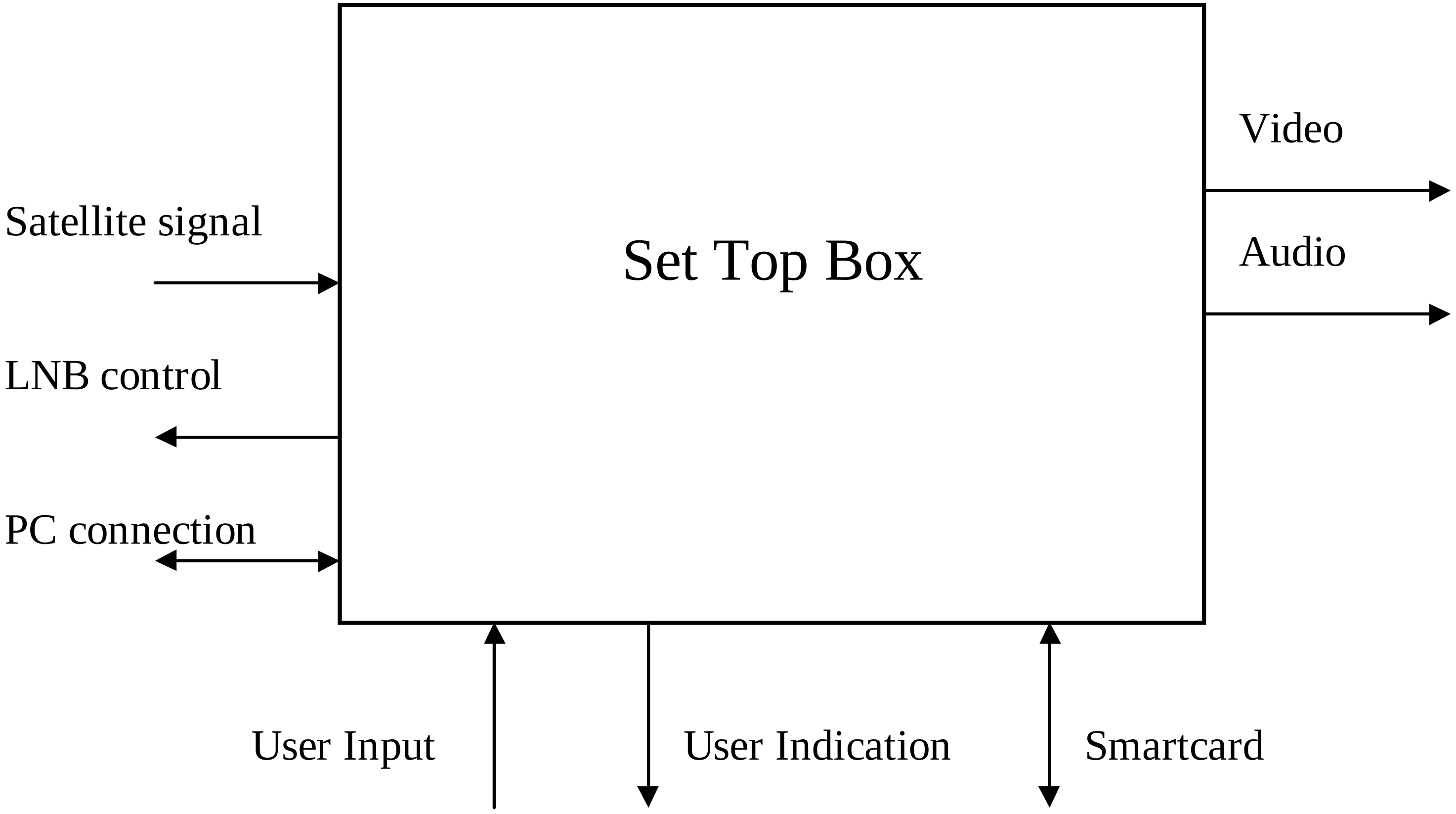
## Overview of execution architecture design steps

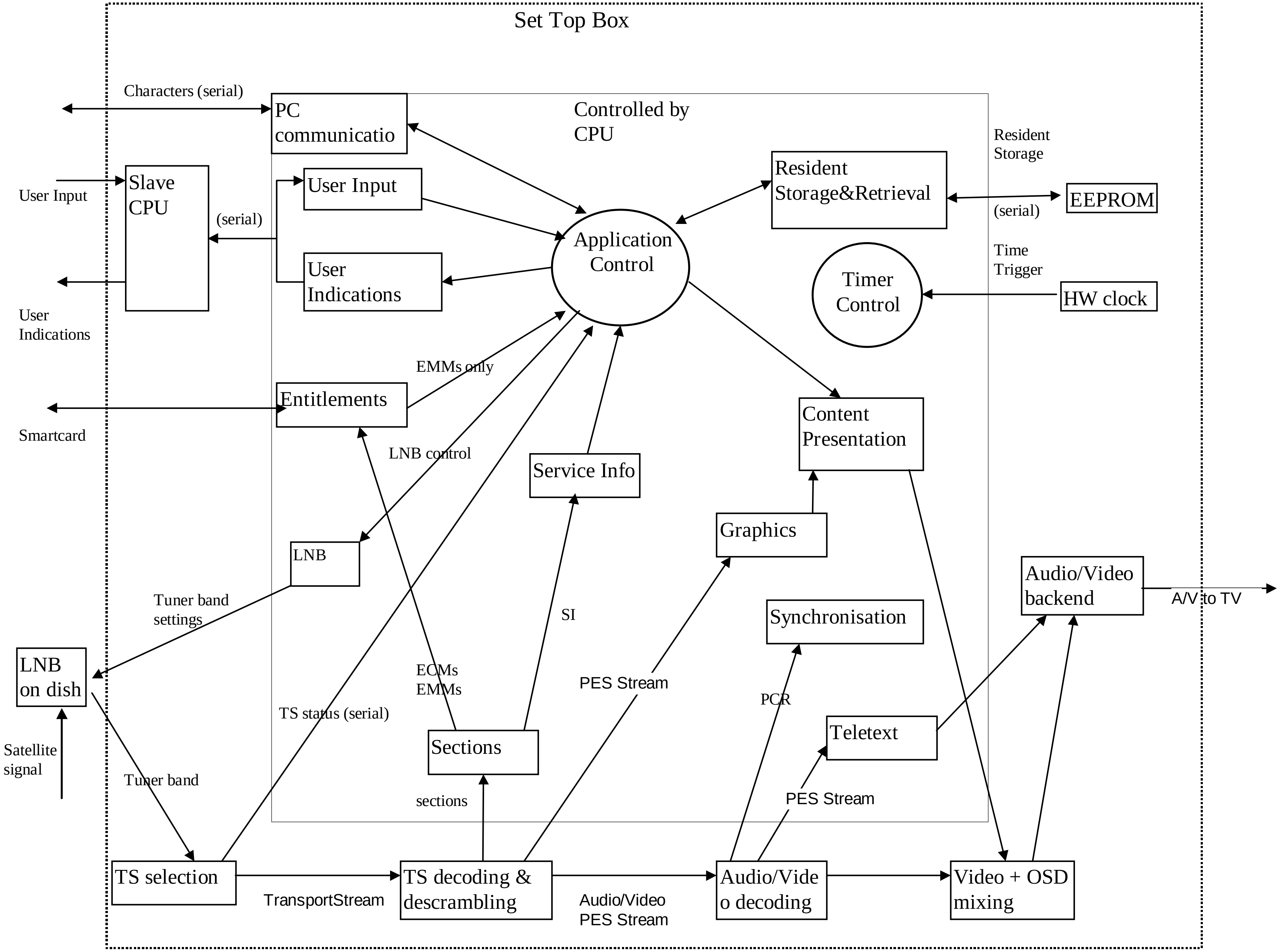
- Get an overview of all external functions / events, and their timing requirements.
  - Select set of critical scenarios
- Structuring: Identify potential active objects.
- Cohesion: Create object to exec unit mapping.
- Task prioritisation: Use RMA estimation.
- Tuning

## Function, event, period, deadline.

- An active function has a body that may run on a different thread, via event redirection.
- An event has a response, that may generate other events (action flow)
- An event has a period
- The response of an event has a deadline (<= deadline)

## Overview of events (1/2)




Set Top Box
Characters (serial)
PC communicatio
Controlled by CPU
Resident Storage
User Input
Slave CPU
User Input
(serial)
Resident Storage&Retrieval
EEPROM
(serial)
Application Control
Time Trigger
User Indications
User Indications
Timer Control
HW clock
EMMs only
Entitlements
Smartcard
LNB control
Content Presentation
Service Info
Graphics
LNB
Audio/Video backend
Tuner band settings
SI
A/V to TV
Synchronisation
LNB on dish
ECMs EMMs
PES Stream
PCR
TS status (serial)
Teletext
Satellite signal
Sections
Tuner band
PES Stream
sections
TS selection
TS decoding & descrambling
Audio/Video decoding
Video + OSD mixing
TransportStream
Audio/Video PES Stream

## Structuring step (1)

- Characteristics:
  - local scope (developers)
  - trade-off : synchronous/asynchronous functions
  - identify active objects
  - one execution unit per active object

# Speaker Notes

Mr. Hassan Gomaa has defined a *design method for concurrent and RT systems*. A reference can be found in the paper and the CTT provides a course for this method. He defines 2 steps.

In the first step a set of  structuring criteria are used to assist the designer in structuring a RT system into concurrent tasks. So these criteria assist in making the trade-off to make a function synchronous or asynchronous. In this step the active objects are identified and all execution parameters are unique.

After this first step, potentially a large number of small tasks are created in the system. This means that the system complexity is high and a lot of task switches occurs, resulting in reducing the performance. In the second step, a set of cohesion criteria are used to assist the overall designer (or architect) in combining a number of execution units together. As such the number of task switches is reduced and so the performance ot the total system is increased. The execution units can be combined easily by letting a number of execution parameters refer to the same execution unit. So a number of objects borrow  the same execution unit without changing the interface.

Note that the first step is done locally by the designers and the second step globally by the architect.

## Structuring step (2)

- GOMAA - CODARTS structuring criteria:
  - I/O:
    - active / passive = interrupt vs polling
    - asynchronous device I/O
    - periodic I/O: different freqs
    - resource monitors: care for consistency

## Structuring step (3)

- GOMAA - CODARTS structuring criteria:
  - Internal:
    - periodic functions
    - asynchronous
    - control (object following a state-transition diagram)
    - user role (“sequential application”)

# Cohesion step (1)

- Characteristics:
  - global scope (architect)
  - reduce task-switching overhead and memory requirements by reducing the number of execution units
  - map active objects on the same execution unit
  - one task per execution unit

## Cohesion step (2)

- GOMAA - CODARTS cohesion criteria:
  - – temporal cohesion (= same priority)
    - different actions from the same event
    - actions with similar periods (when independent)
  - – sequential cohesion (= no interference)
  - – control cohesion (= no interference, exclusive calls)

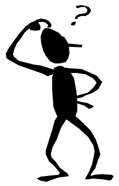
## Tuning step

- Do measurements and RMA(nalysis)
- (only) when the deadline is not met:
  - either the processor is idle now and then, and you can benefit more from concurrency:
    - redo from cohesion onwards
  - otherwize: speed-up critical processing part

## Meeting timing requirements

- Up till now :
  - parametrizable architecture via ‘g4events’
  - used Gomaa to define execution units

  flexibility and orthogonality is achieved
- But what about timing requirements?



# Speaker Notes

Up till now, we have specified a parametrizable execution architecture using the ‘g4event’ model. The usage of this model is based on the method of Mr. Hassan Gomaa during two steps.

These two techniques provides us an execution architecture which is flexible and orthogonal on the functional architecture view.

But the third goal is not yet met : what about guaranteeing timing requirements ??

# Meeting timing requirements (2)

***RATE MONOTONIC ANALYSIS (RMA) :***

- prioritization
  - the shorter the event *deadline*, the higher the task priority of the execution unit
- analyze and guarantee timing requirements
  - FOCUS on *system events and action flow*, not on tasks.

# Speaker Notes

RMA provides design guidelines and real-time analysis techniques that are based on a priority-based pre-emptive scheduling mechanism. Again some references can be found in the paper.

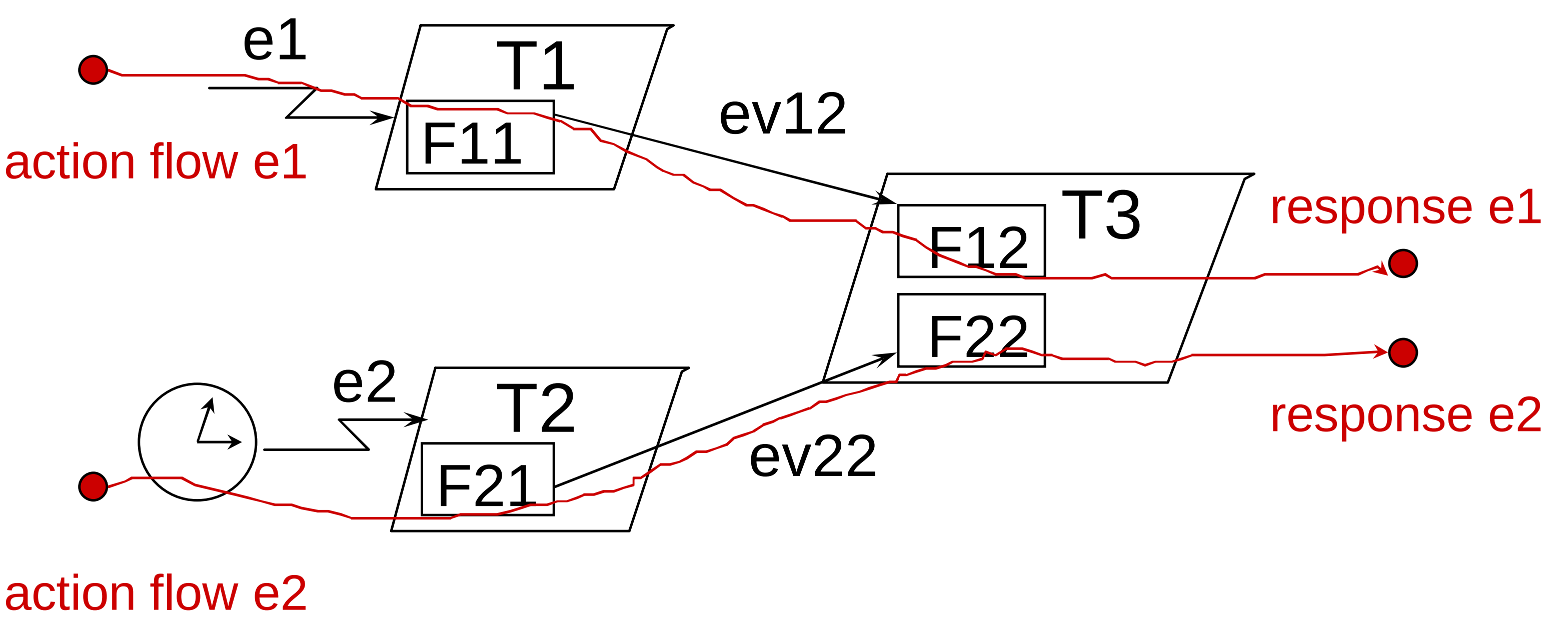
RMA provides guidelines to map execution units on prioritized RTOS-tasks. With the optimal priority assignment, the deadline-monotonic priority assignment, the shorter the event deadline, the higher the task priorities. Since in our model, there is only one task for an execution unit, the terms execution unit and task are synonyms in the context of RMA.

Furthermore RMA provides a technique to analyze and guarantee timing requirements. Note the focus for RMA lies on events and not on tasks.

Let us take a closer look to these guidelines via an example.

# Meeting timing requirements (3)

Sample situation



# Speaker Notes

I used a simplified presentation here :

- objects not shown anymore, only functions.
- indirections not shown anymore, directly the bodies.
- U = tasks (q's not shown)

For a given system, one situation is shown here :

- A system is triggered by events, either external to the system (e.g. hardware trigger e1) or internal (e.g. timer e2), named a system event.
- To respond to a system event, a number of execution units are activated in some order. The communication between these execution units is handled by g4events (ev12 and ev22). This communcation is named the action flow for a certain system event -> red lines.
- Each system-event has a period and a deadline. The period is the minimum distance between two same system-events. The action flow which is triggered by the system event must be completed (response time) before a certain time, named the deadline.

## Real-time scheduling theory, utilization bound

- Set of n tasks with periods T_i, and process time P_i, load u_i = P_i / T_i,
- Schedule is at least possible when tasks are independent and:

$$Load \equiv \sum_i u_i \le n\left(2^{\frac{1}{n}} - 1\right)$$

- 1.00, 0.83, 0.78, 0.76, …. *log 2 = 0.69*.

# Real-time scheduling theory, issues

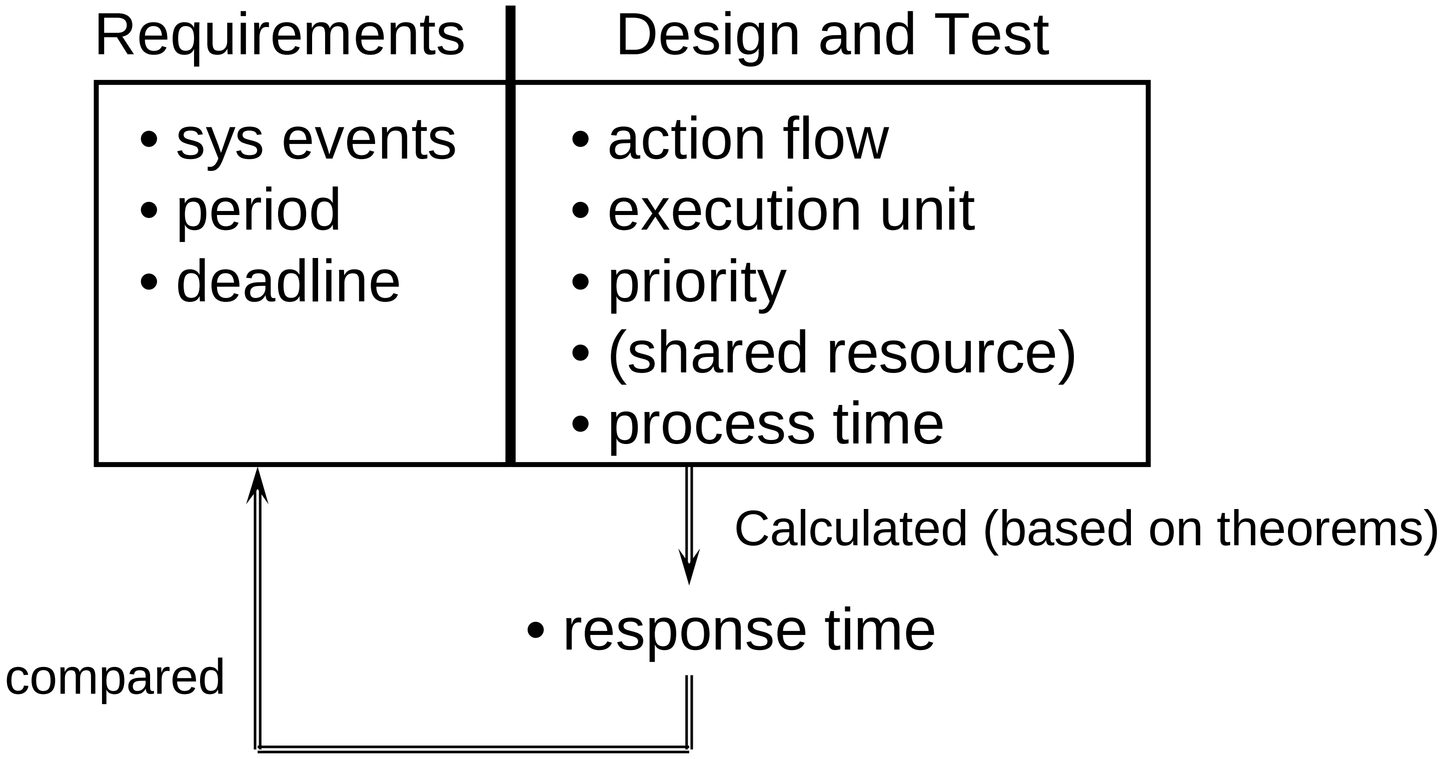
- Note the demand of independence, otherwise extra time must be included in the process times. For example:
- Shared resources (priority inversion) can a snag -> extra parameter, blocking time.
- Support from tools (performax, etc.).

## Meeting timing requirements (4)
## Situation table

| *Specification* | | | *Design & Test* |
|---|---|---|---|
| System event | Period | Dead-line | |
| E1 | 20 | 5 | |
| E2 | 15 | 10 | |

# Meeting timing requirements (5)

## RMA : Situation table



# Speaker Notes

RMA puts the essence of a real-time situation in a Situation Table. This table is filled in by the requirements engineer/ architect. We see two big parts in this table. A first part can be filled in during the requirements phase : system events with their periods and deadlines are captured. The second part describes the action flow, execution units, priorities and shared resources which come out of the design phase. Furthermore process times are measured during testing.

RMA then calculates the response time for each system event, taking into account pre-emption and blocking. This response time is then compared to the deadline and if lower, the timing requirement for that system event is met. RMA further calculates the total CPU usage.

## Results for G+4

- ➔experiences of G+4 STB applying 3 techniques : g4event, Gomaa, RMA
- overhead 'g4events' is neglectable (<1%)
- after 'structuring' (Gomaa) -> 75 execution units
- after 'cohesion' (Gomaa) -> 21 ex. units
- total effort: 3% (analysis & measurement)
  - << MIA analysis & design effort

# Speaker Notes

To conclude, some experiences we had in the G+4 Set Top Box project applying the three techniques : g4event model + Gomaa + RMA.

The ‘g4event’ model provides a parametrizable execution architecture and so supports the goal of being flexible and orthogonal to the functional architecture.

The overhead of the g4events is neglectable: less then 1% regarding a context switch.

The criteria of Gomaa give a good technique of applying the g4event model in the real world. Within the G+4 Set Top Box project, after structuring step we had 75 U and after the cohesion step only 21 U were left.

## Shared resources

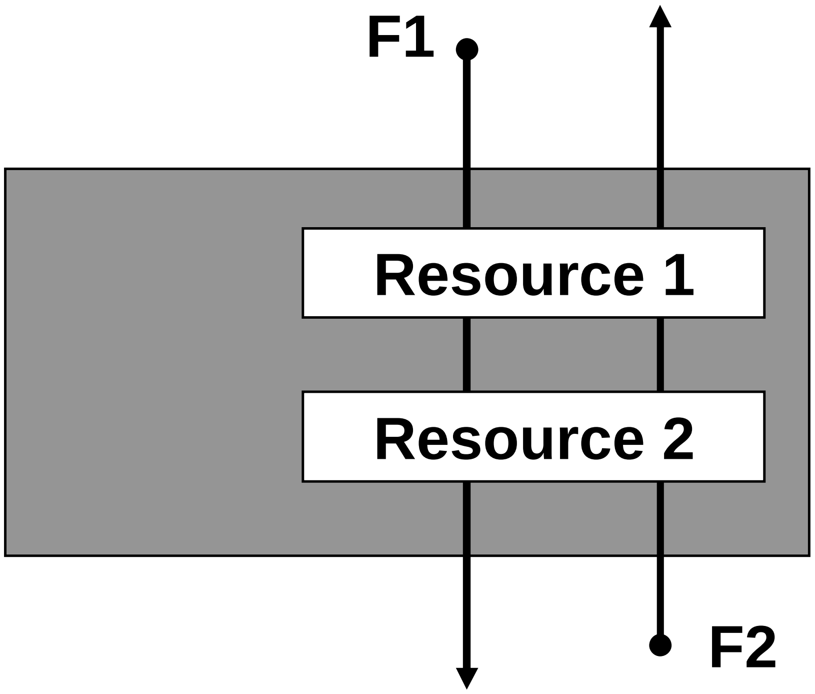
- Example: memory, devices
- How to implement mutual exclusion:
  - disable interrupts (better: partly) or
  - disable task switching (even better: partly)
  - but what about real-time deadlines?
  - even better ...

## Shared resources (2)

- How to implement mutual exclusion (2)
  - semaphores
    - risk of priority inversion
      - ex: small kitchen, bad temper, dishwashing, a fridge
      - solution: priority inheritance
    - use extra “blocking time” in addition to processing time for the relevant events
    - Risk of deadlocks
  - thread decoupling: with “job queue”
    - overhead comparable with semaphores

# Shared resources (3) deadlocks

- Result of mutual exclusion that contain each other.
- Example:



## Shared resources (4) deadlocks prevention

- Order all modules based on their position in the entire system (e.g., logical devices are all placed lower than service modules), based on usage structure (which should form an acyclic graph!)
- Module of order N is only allowed to synchro-nously call methods from modules of order <N
- This means that ‘down’ calls may be synchronous, but ‘up’ calls must be asynchronous (decoupled)

## Shared resources (5) deadlocks prevention

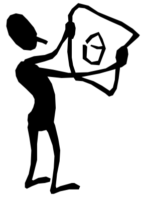
- Absence of deadlocks is guaranteed because semaphores are always passed (locked, ‘P’) in the same order, i.e., the order given by the module ordering
- Now modules can implement their own (local) protection schemes while guaranteeing global absence of deadlocks.
- Yes, a specialized task (thread decoupling) works as well!
- Critical sections must be kept short.

# Priorities: starvation

- Actually this is impossible when applying RMA with hard deadlines.
- However, an example:
  - a monkey sitting on a keyboard

## Summary (1/2)

- identify the real-time situations to be analyzed
- identify system events with Pi & Di
- create situation tables using RMA
  - ⇒ guestimate process times (advanced)
- determine action flows using Gomaa: identify and merge execution units



# Speaker Notes

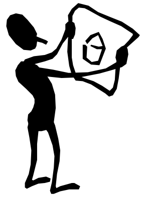
Creating a parametrizable execution architecture is done in several steps during the software development lifecycle. Starting with a mechanism which supports this parametrization, in a first step we analyze the system to be developed on timing requirements and identify the real-time situations which are criticaland are to be analyzed. Note this step is done already in the requirements phase, so very early in the project.

For each situation the system events with their periods and deadlines are identified and a situation table according RMA is created. Note that until here no design has to be created yet. Already process times can be guestimated and a first analyze can be covered, f.I. to discuss with your product management.

In the design phase we determine the actions flows for each system event using the criteria of Gomaa in two steps : identifying and merging the execution units using the parametrizable ‘g4event’ model.

# Summary (2/2)

- identify shared objects, prevent deadlocks with global analysis, rules
- determine deadline monotonic priorities
  - ⇒ estime process times (advanced)
- measure process times
- calculate response times and compare (use a tool)
- corrective actions and iterate



# Speaker Notes

Together with applying the Gomaa criteria we identify the shared data objects. These objects can cause blocking times for RMA.

We then continue with determining the deadline monotonic priorities for each execution unit.

The only missing issue in the situation table of RMA is the process time. This process time can now be estimated and again an analyze can cover a more detailled view on the timing requirements. Out of this analysis we can already steer problems during the design and implementation phase, so before the test phase.

Then we measure process times and fill them in the situation table. RMA then calculates the response times for the system events and compares them to their deadlines.

We can do corrective actions if deadlines are missed and iterate this process again by adapting the situation table according the new  implementation.

## Conclusions and remarks (1/3)

- Gomaa and g4events provide hierarchy in work (local versus global)
- performance improvements without large changes
- performance predictions with RMA are useful
- Lower SW levels: true real-time
- Deadlocks have been a major problem

# Speaker Notes

Gomaa and the g4event provide a flexible way to design an execution architecture. In the first step designers can work locally and in the second step the architect works on a global base by applying the techniques provided by the g4event model and Gomaa.

Performance improvements could be done easily by changing some parameters of the g4event model.

Preformance predictions are useful using RMA. Several alternatives can be analyzed without implementing them, so that choosing between these alternatives becomes easy.

Applying the 3 techniques gives a stable process throughout the software development lifecycle, but starting already very early in the project gives you an even more stable process so that you detect earlier missing timing requirements or requirements which can’t be met to negotiate with your product management.

## Conclusions and remarks (2/3)

- Stable process (management context)
- Start during requirements phase
- Be pragmatic and don’t overdue
- The understanding counts, not the administration or tools (but do document)

## Speaker Notes

I would like to end with thanking the IST (information and software technology centre) for their support using Gomaa and in specific RMA to be very useful techniques in creating a parametrizable execution architecture.

For those of you who want to know more about experiences using RMA in projects, there is a special interest group this evening covering the Rate Monotonic Analysis in more detail together with a market place where you can experiment with the technique. So check in for SIG8.

Thank you.

## Conclusions and remarks (3/3)

- Thanks to IST for support (L. Steffens)
- Note that interrupt routines can be modelled as independent tasks as well!
- LBNL: Focus on requirements, events, and actionflow, not on tasks or threads.