COMPARE

Component-based approach for real-time and embedded systems

vincent.seignole@fr.thalesgroup.com

http://www.ist-compare.org

COMPonent Approach for Real-time and Embedded

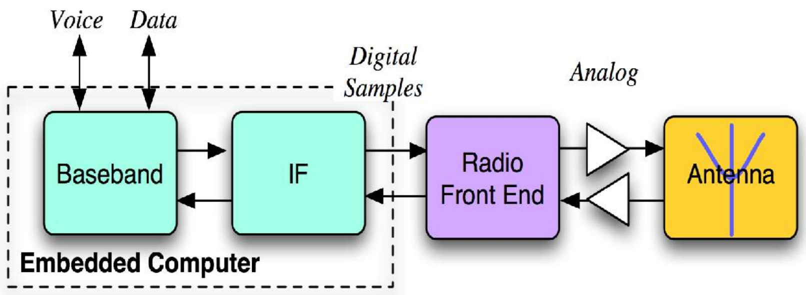
Who is COMPARE

- Collaborative european IST project (running until end 2006)
- Focus : *component-based approach for real-time and embedded systems*
- Project consortium:

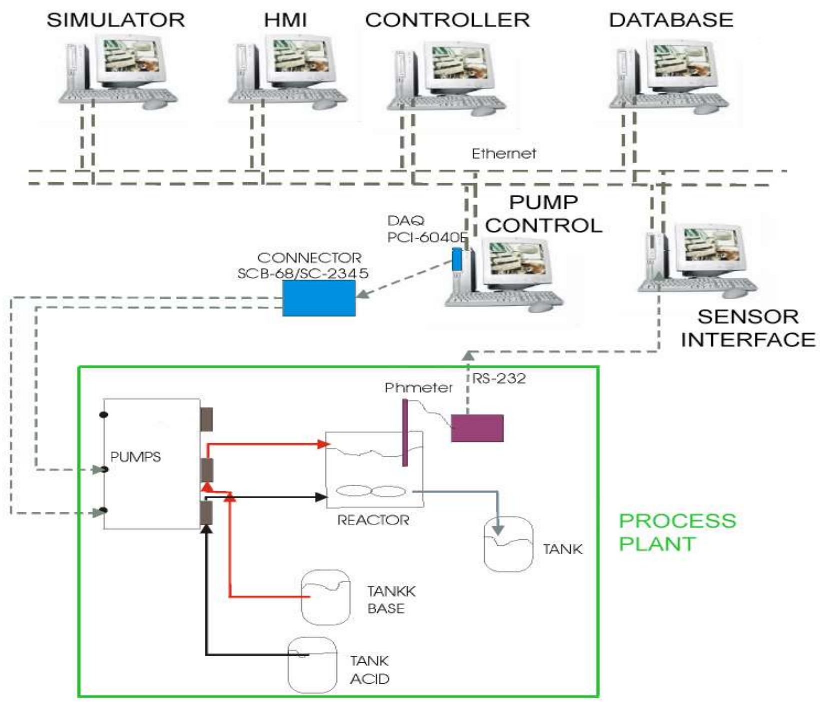
COMPARE

targeted applications

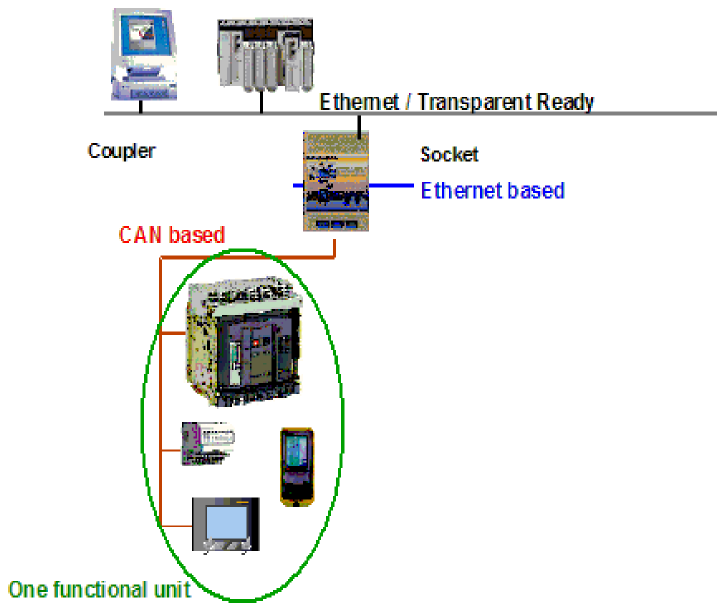
- *Software defined radio*



- *distributed control systems*



- *Electrical distribution*

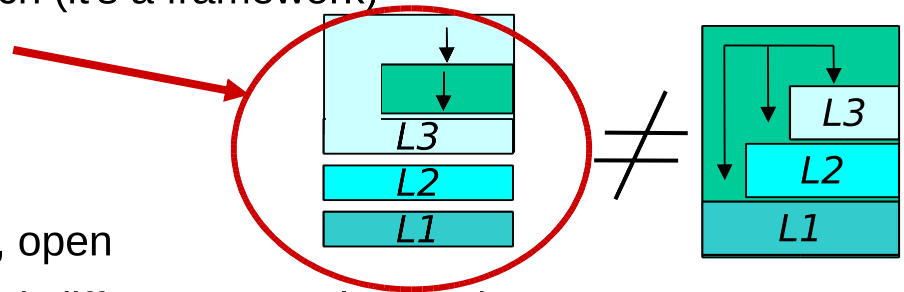


\+ *automotive systems*

Introduction / Motivations

- RT/E systems sw-engineering via component approach
  - ⇒ Motivations
    - Designing real-time systems like communication ones is hard
      - → **real-time** constraints (schedule, synchronization), **embedded** (code adapted to hardware target **platform** ?)
      - → hard **integration** times: increased complexity, more functionalities, more variability
      - → **openness** constraints (e.g Software defined radio)
    - Reusing previous developments in slightly different scopes can still be difficult
      - → example : portability on different RTOSes
    - Reuse (when made possible) granularity is traditionally too coarse
  - ⇒ Relevance of Component based approach:
    - fosters **reusability** of system functions in the scope of product lines and beyond
    - Resulting system sound architecture, eases analysis of system
    - **portability**: allocation of components to different kind of resources without effort
    - Offers a pattern for a clean **separation of concerns**
  - ⇒ **But** today, no broadly applicable component approach for real-time and embedded systems is available

- What we propose
  - ⇒ New approach for sw architecture of real-time embedded systems
    - Definition of a component model and accompanying toolset
      - → following a smartly layered approach (it's a framework)
      - → enforcing separation of concerns
        - e.g real-time scheduling / algorithmic code
      - → Lighweight, versatile, value-added, open
        - reuse component across (truly) different execution environments
        - ability to embed nothing more than what needed in the infrastructure
        - coming with a collection of pre-existing services and architectural patterns
  - ⇒ Builds on / adapts / profiles existing standards (OMG Lw-CCM)

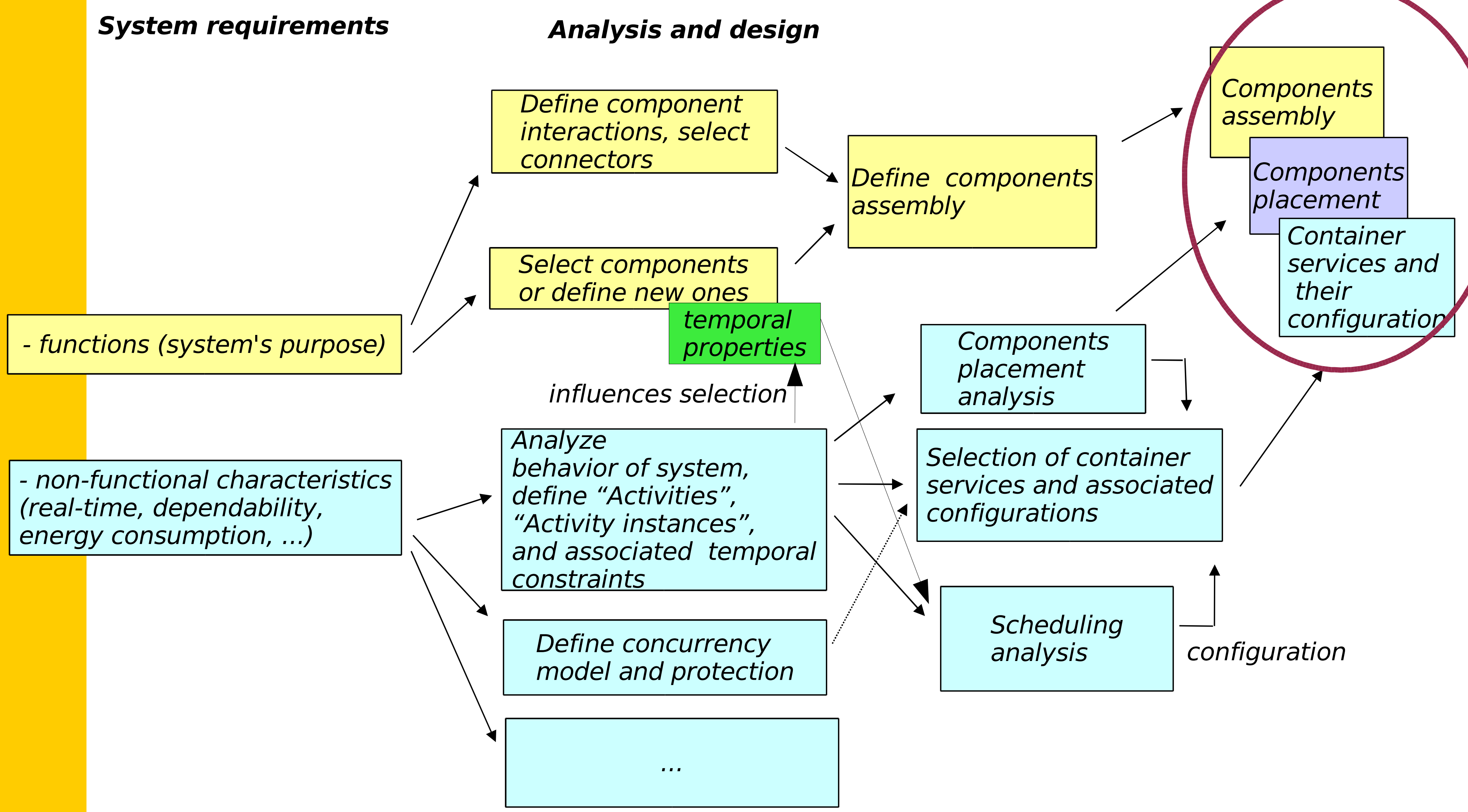


- This talk concentrates on:
  - ⇒ open container supporting real-time features
  - ⇒ reusable interactions support

Precisions: some key requirements

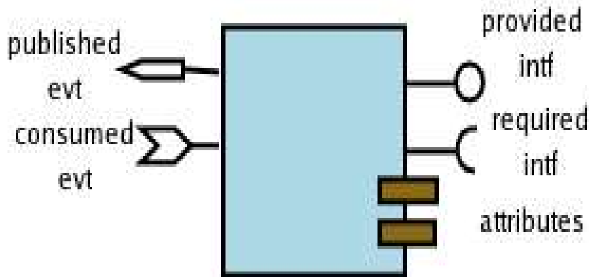
- Follow and extend **standard** approaches
  - OMG lightweight CCM, OMG D&C specification, QoS for CCM
  - push the obtained extensions for standardization
- **Separation of concerns**
  - ultimately reusable software versus embedding it into a variety of non-functional environments (real-time platforms, etc ...)
- Deal with **real-time**
  - integrate basic real-time ingredients, scheduling strategies choices, concurrency management and realization transparently to components code
- Allow smooth integration of / gatewaying to **legacy** code / systems
  - crucial for acceptance in an industrial context
- Examine carefully **tradeoffs** in terms of **tailoring** / profiling standards as well as implementation techniques to get optimal performance and footprint
- Realize **implementations** of the fwk with following underlying RT platforms:
  - real-time CORBA (e*ORB) on VxWorks / PPC and OSE ck / TI C55 DSP (SDR)
  - on OSEK (automotive domain static RTOS) + OSEK Com (Electrical breaker)

supported approach – separation of concerns

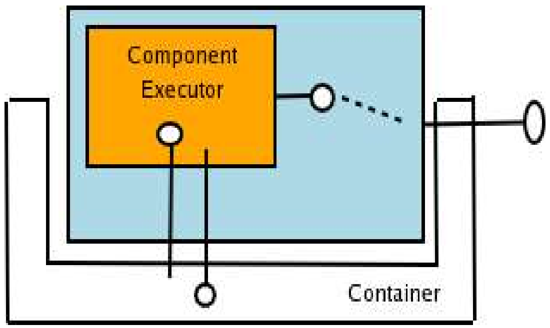


Lw-CCM basics (1/2)

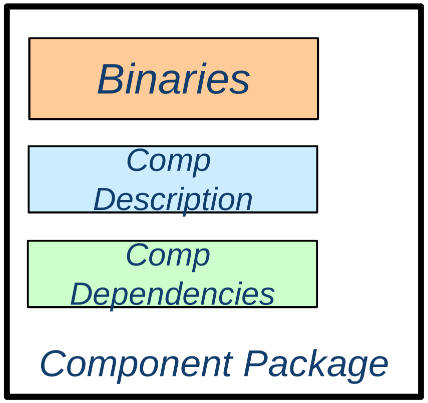
- A component model
  - ⇒ description of component types
    - provided / required services
    - events published / consumed



- An execution environment
  - ⇒ containers
    - shield components developers from middleware technical concerns
    - Underlying CORBA middleware support
      - → but not only (in our extensions)



- A packaging model
  - ⇒ software-package format
  - ⇒ self-descriptive : contents, dependencies



- A deployment infrastructure
  - ⇒ assembly descriptors + runtime support

Lw-CCM basics – how to build a component (2/2)

IDL definitions (interfaces, ...)

refers to

Component types and home types definitions (IDL3)

```
component C
{
  uses I the_i;
  provides J the_j;
 attribute string name;
};

home C_home manages C
{
};
```

Component implementation framework (CIF)

```
local interface CCM_J : J { }

local interface CCM_C_Context
{
  I get_connection_the_i();
};

local interface CCM_C : EnterpriseComponent
{
   CCM_J get_the_j();
  attribute string name;
};

local interface CCM_C_home { ... };
```

IDL2 equivalent definitions

```
interface C : Components::CCMObject
{
   attribute string name;
};

interface C_homeImplicit
{
    C create() ...
};

interface C_home : C_homeImplicit ...
{
};
```

Implementation by developer (also called “executor code”)

component binary

implem delegates to

(CORBA) Implementation of these interfaces is automatically generated

This is not the container, it is the “component envelope”

Enhanced interactions – architectural patterns (1/5)

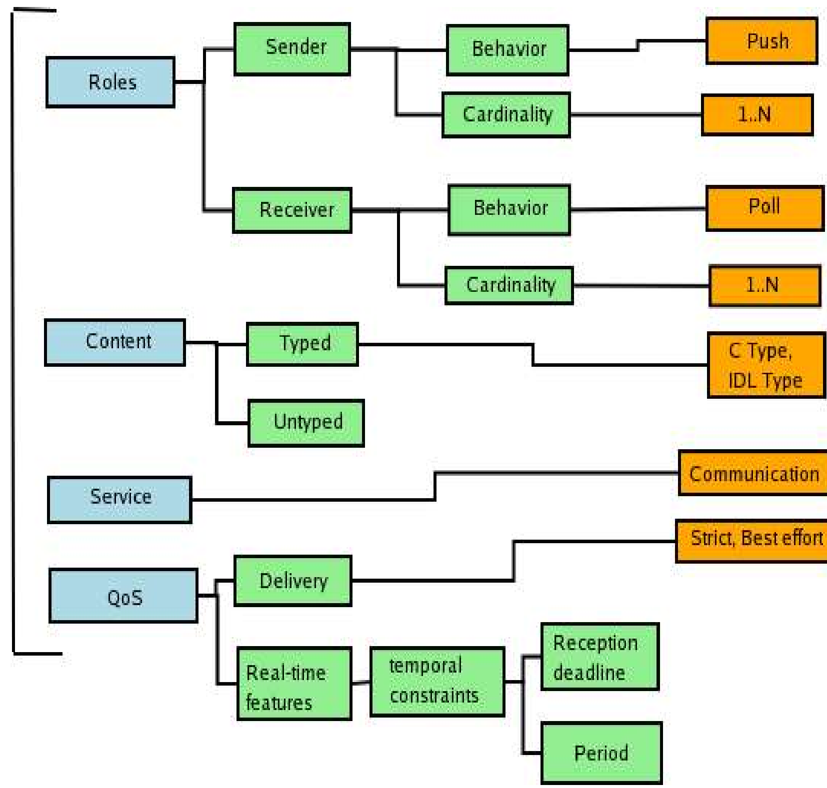
Extension of CCM capabilities in terms of interactions

- ⇒ Currently :
  - ☐ Synchronous method invocation
  - ☐ (1,n) push-push typed events propagation

Without any QoS

- ⇒ Other architectural styles of interest in real-time systems
  - ☐ Pipes and filters
    - → classical in communication protocols domain
  - ☐ Variants of publish-subscribe
    - → push/pull, typed / untyped events
  - ☐ Blackboard pattern
- ⇒ Characterization of an interaction via
  - → Participants to the interaction
    - ☆ roles, and cardinalities
  - → Type of service
    - ☆ communication or/and synchronization
  - → QoS features
    - ☆ synchronicity, dynamic behaviour

example



Enhanced interactions (2/5)

- Introduced
  - ⇒ the <u>Connector</u> concept as in the ADLs to abstract interactions
  - ⇒ Mechanisms to provide Connector implementations
    - ability to define new port types (in addition to : provides, uses, consumes, ...)
    - ability to define new connector types
    - rules for connectors implementation and packaging
  - ⇒ Extensions of application descriptions
  - ⇒ Extensions to infrastructure to deal with connectors
- We have a set of classical connectors available:
  - Deferred synchronous calls (callback based, poll based)
  - Event bus mechanisms
  - Optimized direct method calls
- We can provide easily(w/o modifying the obtained comp. model):
  - blackboard connector
  - connectors for ad-hoc messaging scheme / ad-hoc transport (i.e non CORBA based)
  - streaming connector

Enhanced interactions (3/5)

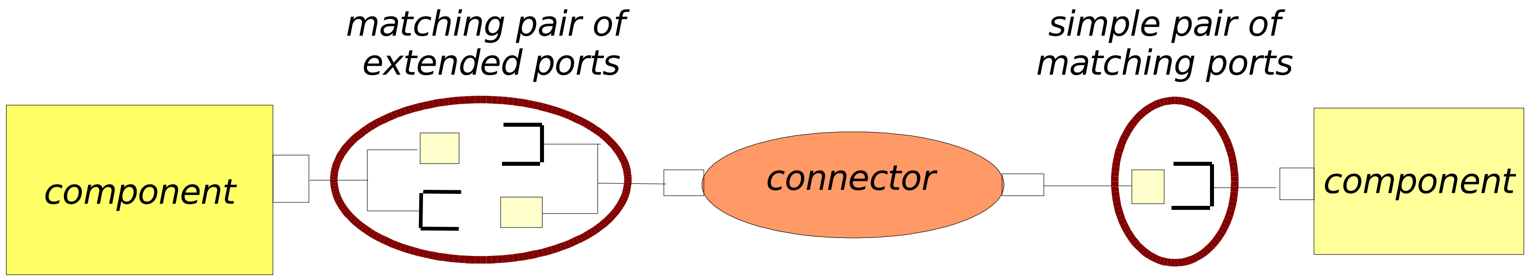
Component / Connector model :

⇒ "Extended port concept":

- can define the port type needed in case of specific interaction model.
- can be parameterized by an interface type or event type (like C++ templates).
- extended port as collection of needed / used interfaces + associated semantics
  - → Component model becomes a little bit closer to UML2 component model

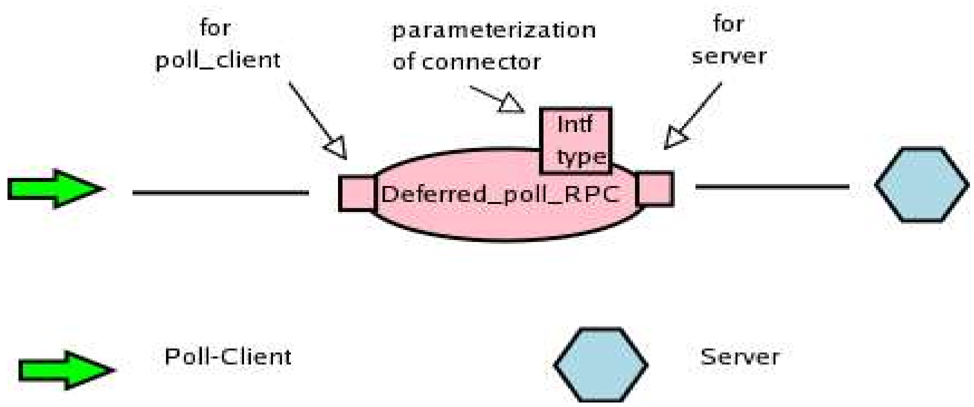
⇒ Connector concept :

- stands for interaction entity having some "extended ports" as well as attributes
  - → looks like components but see some slides later for differences



Enhanced interactions – example (4/5)

- Deferred synchronous call with polling



- Client uses an interface derived from I by transformation:
  - namely RPC_Poll_I
    - call foo( ) passing arg,
    - does not block
    - obtains “long_Poller” to check for result later

```
Port_type RPC_Poll_Client<I>
{
  provides trsf(I) client_port;
};

connector RPC_Poll<I> {
 provides_port RPC_Poll_Client<I> fpc;
 uses I for_server;
};
```

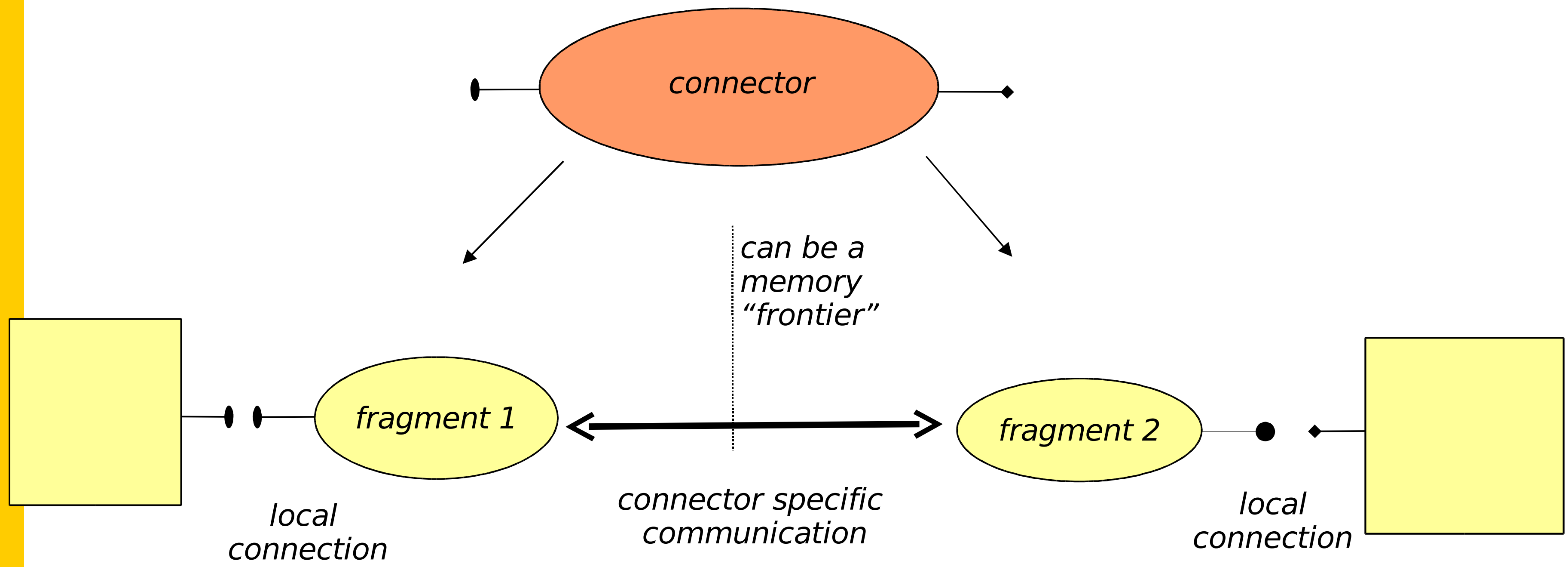
```
component Client_component {
  uses_port RPC_Poll_Client<I> my_port;
};

component Server_component {
  provides I a_port;
};
```

```
interface I {
   long foo( in string s );
 };
local interface long_Poller {
   bool is_available();
   void get_result( out long result );
   void check_for_exception( … );
};

// interface effectively used by caller
// (refers to trsf(I) above)

local interface RPC_Poll_I {
   long_Poller foo( in string s );
};
```

Enhanced interactions (5/5)

Connectors implementation :

- ⇒ is naturally "fragmented" (connector parts co-located with components involved in interaction)
- ⇒ connector fragments implementations follow the component approach
  - ☐ they implement the "IDL3 extended ports" concept



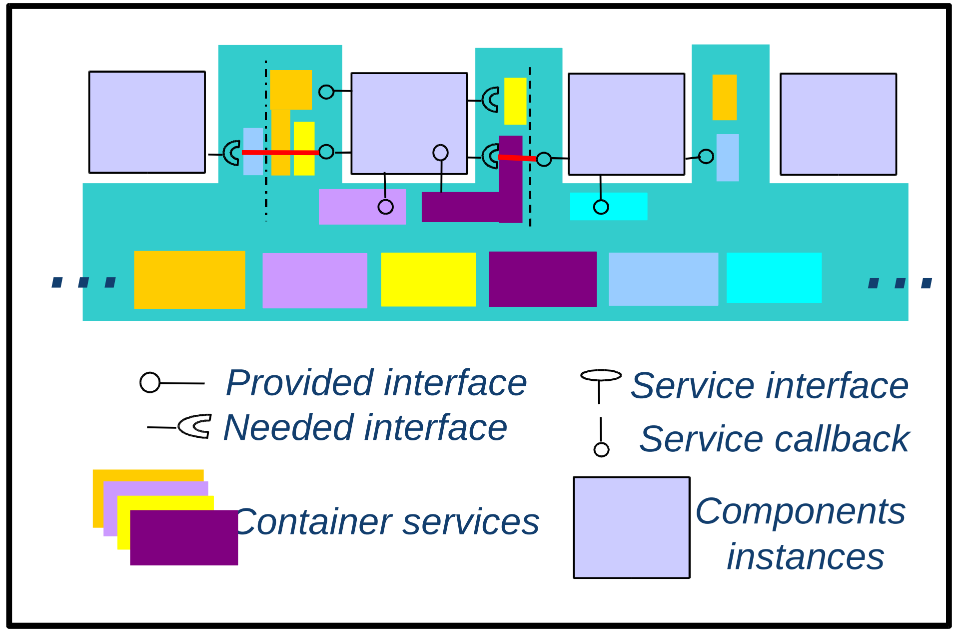
Open container (1/3)

Definition of an open (extensible) container to support various combinations of **plug-ins** for **non-functional** aspects handling.

- Motivations to provide a container:
  - Provide true **separation of concerns**
    - Means in practice: code no more polluted with tasks creations, synchronization, …
      - i.e. these aspects are to be integrated transparently to component code and described (not hard-coded).
    - This is what Containers (in component technologies) are meant for!
  - Allow creation of an **open** and **customizable** execution environment via containers
    - To fit exactly application needs (in terms of threading, communication, *etc.*)
    - Coming with a set of pre-existing services (e.g scheduling, thread-pools,memory pools, …)
- Approach:
  - Build custom containers starting from modules: called ***container services***
  - Define generic interfaces to implement container-integrable modules

Open container (2/3)

- Container as host for:
  - → homes, components
  - → service plug-ins: are aggregation of objects involved at well-specified occasions (corresponding to "integration points" below) in the container
- provides what we call "<u>integration points</u>" for services:
  - → requests *<u>interception</u>*
  - → control at *<u>components creation time</u>*
  - → control at *<u>components connection time</u>*
  - → integration of *<u>connector fragments</u>*
- APIs (1) are defined for insertion of service at "integration points"
  - → see next slides for details
- API defined for Service plug-ins
  - → Service plug-ins install objects implementing interfaces from (1) at "service integration points"



Open container (3/3)

- Integration points:

Pre and Post - request interception	interceptor : bound to comp instance + method (smart proxies techniques) access to request (end-to-end) context other granularities possible
Request interception – In place of	Corresponds to integration of Connector fragments
components creation time and servants activation / deactivation time	activating the component servants on a POA (having specific policies) and in a specific manner
Container internal interfaces (component context)	for container services interfaces needed to be presented to component
Components Callbacks	interfaces implemented by component that the container uses

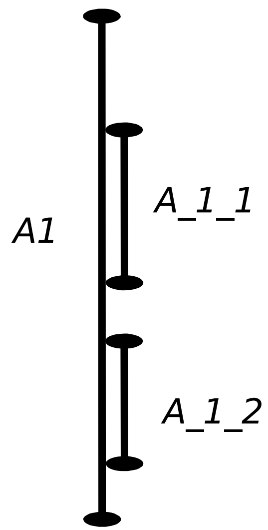
Real-time – model of computation (1/2)

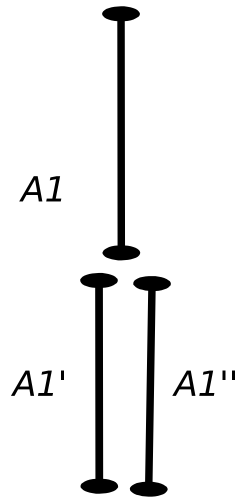
Notion of “Activity” as a design concept:

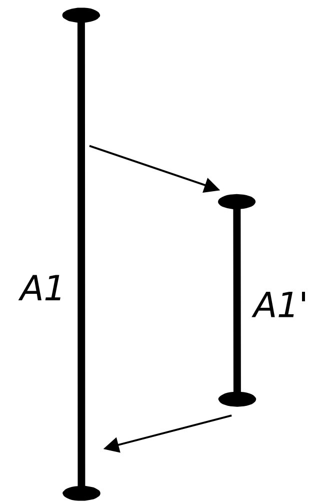
- ⟹ builds on the notion of “distributable” thread of RT-CORBA 2
- ⟹ Activity : logical entity corresponding to a behavior of the system
  - can be described as a sequence of actions whose starting point can be identified, as well as stop point (which can be implicit)
  - associated to a component instance, a facet, and a method
- ⟹ Activity instance :
  - realization of an Activity at particular (absolute) time, under a particular context.
- ⟹ Real-time formal vocabulary
  - Classical concepts:
    - → release time, relative deadline, jitter, importance
  - as annotation of activity instances

Real-time – model of computation (2/2)

Activity compositions patterns (can be combined)







nesting of Activities

A1, A_1_1, and A_1_2 can be annotated with RT-constraints

Typically : situation arises in the case of nested component calls

chained transfer of activity

A1' and A1'' start points is A1 end point.
Release time of A1' and A1'' = end time of A1

Typically : situation arises as natural precedence constraints, A1 not triggering directly A1' and A1''

fork and join or fork and forget

A1' starts at specific component interaction

Mapping of activities

- Container services identified and considered:
  - ⇒ realize the activity patterns from previous slide
    - activity creation,
      - → timers, interaction with component wrapping hardware
    - nested activity creation
      - → specific components interaction (as part as “specific” activities)
    - new (forked) activity
      - → same
    - transfer of activity
- Ingredients / container mechanisms to deal with “Activities”
  - end-to-end request context (carrying “Activity context” tailored for app)
  - Pre and post interception
  - combined with synchronous, deferred synchronous, async connectors

Integration of “activities” and real-time CORBA 1.0

- Real-time CORBA 1.0 :
  - ⇒ is a priority based real-time middleware
- Schedulability analysis outputs:
  - ⇒ set of priorities for
    - activities, nested activities, forked activities, and transferred activities
- Container services identified and involved:
  - ⇒ RT-CORBA Client propagated policy application
    - usage of an “On activation” integration point
  - ⇒ Application of the set of priorities:
    - pre or post – interceptors (coming with Activity mgt plugins from previous slide):
      - → configured with mapping : activity context to priority
      - → using the RT-CORBA API to set priorities (i.e RTCORBA::Current)

Mapping can also be done onto CORBA dynamic scheduling and deadline schedulers.

Can also be done onto other mechanisms like typically direct POSIX primitives

Concurrency management

Mechanisms to automate management of concurrency:

- ⇒ Intent: manage as far as possible concurrent accesses protection without presenting Lock APIs to the component developer.
- ⇒ Some ideas and considered techniques

  - Lock-based technique :
    - → lock protections at component instance level, facet level, ...
    - → done by container plug-in : lock / unlock with pre / post interception
    - → placement of locks is not easy, too coarse granularity
      - but description of app activities can help to make the analysis.

  - Alternatives (under investigation):
    - → implement “Transactional software memory” patterns
      - can be automated to some extent
    - → directly use “lock-free” data structures when possible
      - responsibility put on component developer.

Ongoing and future work

- Assessment of the container and real-time techniques considered
- Implementation of framework:
  - ⇒ onto e*ORB SDR C++ onto VxWorks / PowerPC
  - ⇒ onto e*ORB SDR C onto OSE compact kernel on TI C55x DSP
  - ⇒ minimalistic implementation onto OSEK / automotive in EC++
    - need to optimize very much and allow a fully static approach
- Implementation of test beds to validate the whole approach:
  - ⇒ Software defined radio waveform (THALES)
  - ⇒ Electrical breaker (Schneider-Electric)

Conclusion

- More information and details at:
  - ⇒ **http://www.ist-compare.org**

- Thank you for your attention