

Ada Europe 2015

# A Task-Based Concurrency Scheme for Executing Component-Based Applications

*Francisco Sánchez-Ledesma*

*Juan Pastor*

*Diego Alonso*

*Bárbara Álvarez*



Universidad  
Politécnica  
de Cartagena



**DSiE**  
DIVISIÓN DE SISTEMAS E  
INGENIERÍA ELECTRÓNICA



# Contents

- Technological context: **CBSD, MDSD, Cforge Tool Chain.**
- General Approach: **Solution drivers, the WCOMM component metamodel.**
- Task Based-Concurrency Scheme.
- Execution Model.
- Example.
- Deployment and Analysis.
- Conclusions and further work.

# Tech. Context: CBSD + MDSD

## ➤ **CBSD** *Component Based Software Development*

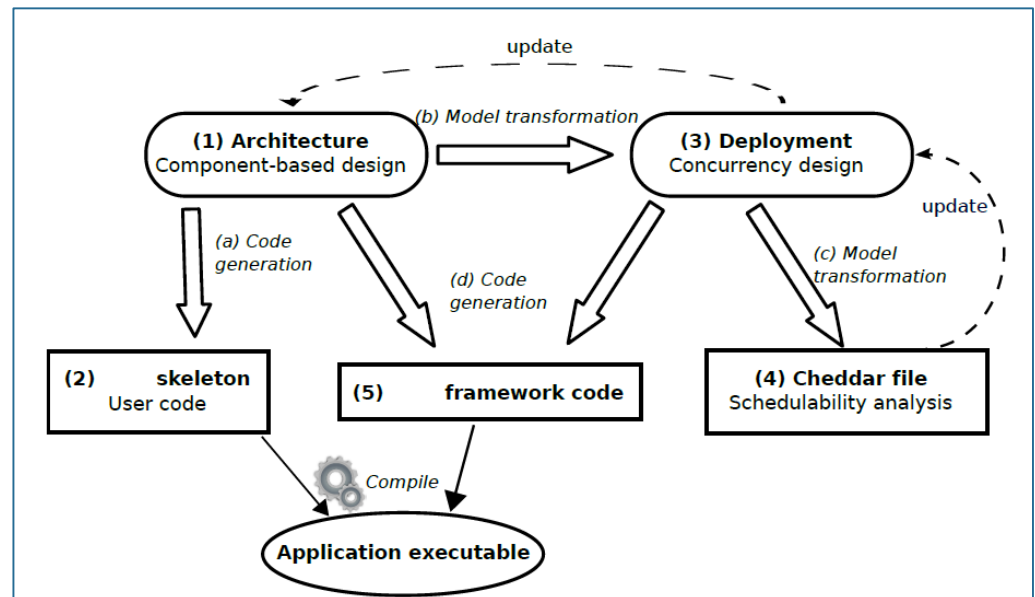
- **Architectural software components:** self contained units that encapsulate their state and behaviour, that exchange typed messages only through their ports, and that have only explicit context dependencies.

## ➤ **MDSD** *Model Driven Software Development*

- **Meta-models, models** and, ...
- **Transformations:** how models conforming to a meta-model are translated to models conforming to other meta-models or to code.

## ➤ **C-Forge** *Tool Chain*

- **WCOMM** component model.
- **FraCC** execution framework.



# Tech. Context: CBSD + MDSD

- We are using **CBSD** to design/implement applications.
- We are assuming a component model (yet another) ...
- ... and we need to **link app model** to an **executable implementation** ...

## Design concepts.

- Architectural units.
- State-charts, orthogonal regions.
- Ports and messages.

- ↑ Very suitable for application construction.
- ↓ Hinder performance analysis schedulability.

## MDSD

to the rescue ...



**How to map the concepts?**

## Execution model concepts.

- Nodes, processes, threads, tasks.
- Synchronization primitives ...
- Functions, objects, methods ...

- ↑ Directly related to performance analysis.
- ↓ Less suitable for application construction (more low level details)

# Tech. Context: CBSD + MDSD

## Design concepts.

- Architectural units.
- State-charts, orthogonal regions.
- Ports and messages.

## MDSD

to the rescue ...

*How to map the concepts?*

## Execution model concepts.

- Nodes, processes, threads, tasks.
- Synchronization primitives ...
- Functions, objects, methods ...

**Components** are **passive entities** invoked sequentially by a single threaded run time.

Cyclic Executive  
Predictable, but ... rigid

**Components** are translated to **processes** and a **middleware** is used for message exchange

Flexible, but penalises performance and hinder the analysis.

## OO framework solutions.

**Components** are translated to **composite objects**.

**Tasks queues and thread pools**  
(Java.util.concurrent, std::asynch  
C++11, IOS Great Dispatcher...)

Flexible, powerful, expressive, but thought to increase throughput and productivity, not to ensure predictability.

# General Approach. Solution drivers

## OO framework solutions.

- Components to composite objects.
- Tasks queues and thread pools

Flexible, powerful, expressive, but designed to increase throughput, not predictability.

## **This is the starting point, but with some extra-requirements:**

- The number of resulting threads, as well as their timing properties must be known, so that schedulability analysis can be performed.
- The timing properties must be present in the component model, so the concurrency model can be derived from the component model.
- Early testing of different deployments (test-driven deployment).
- Possibility of dynamic reconfiguration of deployment depending on current computational resources and computational load.

# General Approach. Solution drivers

... some extra-requirements ...

**Developer must control application deployment in nodes, processes and threads.**

- Let the developer decide how many (workers) threads execute the application.
- Make the computational load of worker threads static.

The computational load is decided by developer before execution instead of by the system at execution time.

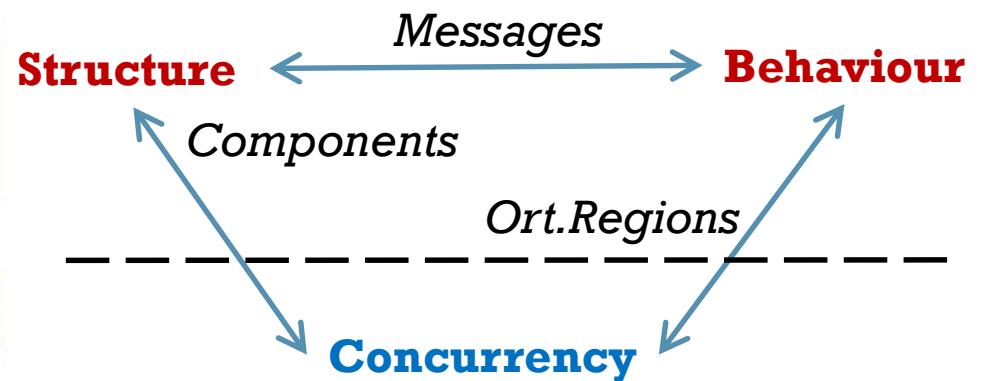
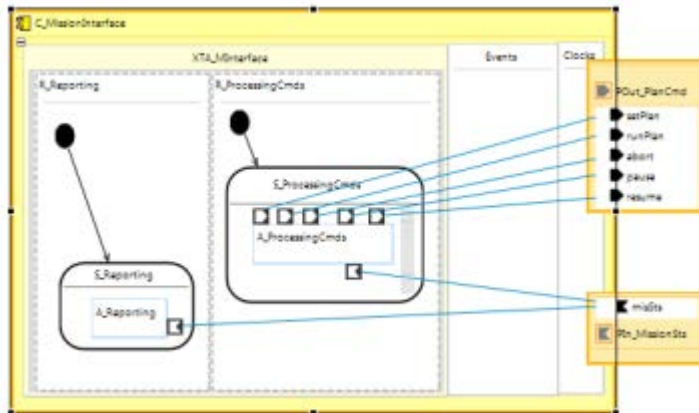
- Create a cyclic executive inside each thread in order to schedule region execution.

# General Approach. The WCOMM Component

- **White-box software units** that encapsulate their **behaviour**.
- Communicate by **sending messages** to each other only through their **compatible ports**
- Messages are grouped into interfaces, and follow the **asynchronous without response** scheme

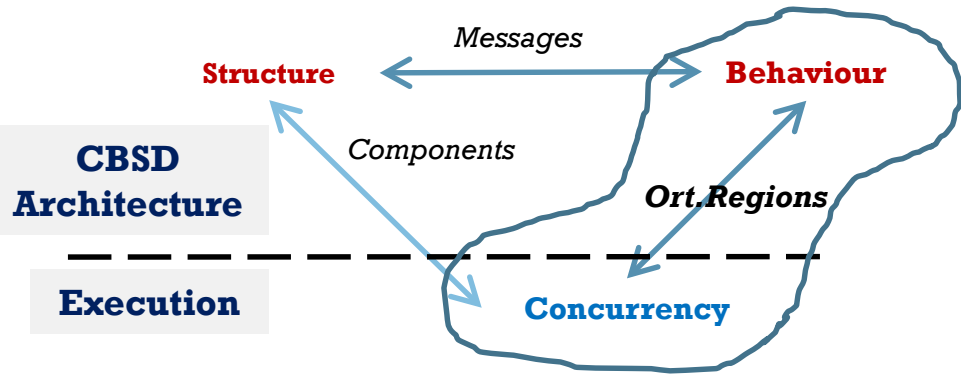
**Ports** are **flow ports**:

- **non-atomic** (messages can have parameters of any type)
- **bi-directional** (thanks to protocols),
- **behavioural** (messages can fire events in timed automata)





# Task Based Concurrency Scheme



**Regions** are the **link** between architecture and execution.

**Regions** contain the **activities** which must be executed by the component depending on its internal state.

**Application:** Set of **components**

$$\text{App} = \{ K_1, \dots, K_N \}$$

**Components:** Set of **Orth. Regions**

$$K_i = \{ R_{i1}, \dots, R_{ir} \}$$

**Regions:** Set of **States** (and transitions)

$$R_{ij} = \{ St_{ij1}, \dots, St_{ijs} \}$$

**States:** Execute **ONE Activity** or none

**Activities** model **sequential tasks**

**Orthogonal region timing properties.**

$$T_{reg}^i = \gcd(T_{act} \in R^i)$$

$$WCET_{reg}^i = \max(WCET_{act} \in R^i)$$

$$CL_{reg}^i = \max(CL_{act} \in R^i)$$

**Tasks** have **timing requirements**: Period, minimum inter-arrival time, deadline, worst execution time, ..., criticality, ....

$$St = \{ T_{act}, WCET_{act}, CL_{act} \}$$

# Execution Model

## Orthogonal region timing properties.

$$T_{reg}^i = gcd(T_{act} \in R^i)$$

$$WCET_{reg}^i = max(WCET_{act} \in R^i)$$

$$CL_{reg}^i = max(CL_{act} \in R^i)$$

**Application** is executed in a set of **nodes**.

$$App = \{ N_1, \dots, N_M \}$$

**Nodes** contain **processes**.

$$N_i = \{ Pr_{i1}, \dots, Pr_{ir} \}$$

**Components** are assigned to **processes**

$$Pr_{ij} \{ K_{ijk} \}$$

**Processes** contain **threads**.

$$Pr_{ij} \{ Th_{ijt} \}$$

**Threads** can be characterized by period, worst execution time and priority band.

$$Th = \{ T_{th}, WCET_{th}, PB_{th} \}$$

## Threads timing properties.

$$T_{th}^i = gcd(T_{reg} \in Th^i)$$

$$WCET_{th}^i = \sum (WCET_{reg}^i \in Th^i)$$

$$PB_{th}^i = max(CL_{reg} \in Th^i)$$

**Threads properties** can be derived from the assigned regions.

**Regions of a process' component** can be assigned to any of these threads providing thread's PB is compatible to region's CL.

# Execution Model. Mixed Criticality.

## Orthogonal region timing properties.

$$T_{reg}^i = gcd(T_{act} \in R^i)$$

$$WCET_{reg}^i = max(WCET_{act} \in R^i)$$

$$CL_{reg}^i = max(CL_{act} \in R^i)$$

## Threads timing properties.

$$T_{th}^i = gcd(T_{reg} \in Th^i)$$

$$WCET_{th}^i = \sum (WCET_{reg}^i \in Th^i)$$

$$PB_{th}^i = max(CL_{reg} \in Th^i)$$

## Criticality Levels

HL > ML > LL

## Priority Bands

HP > MP > LP

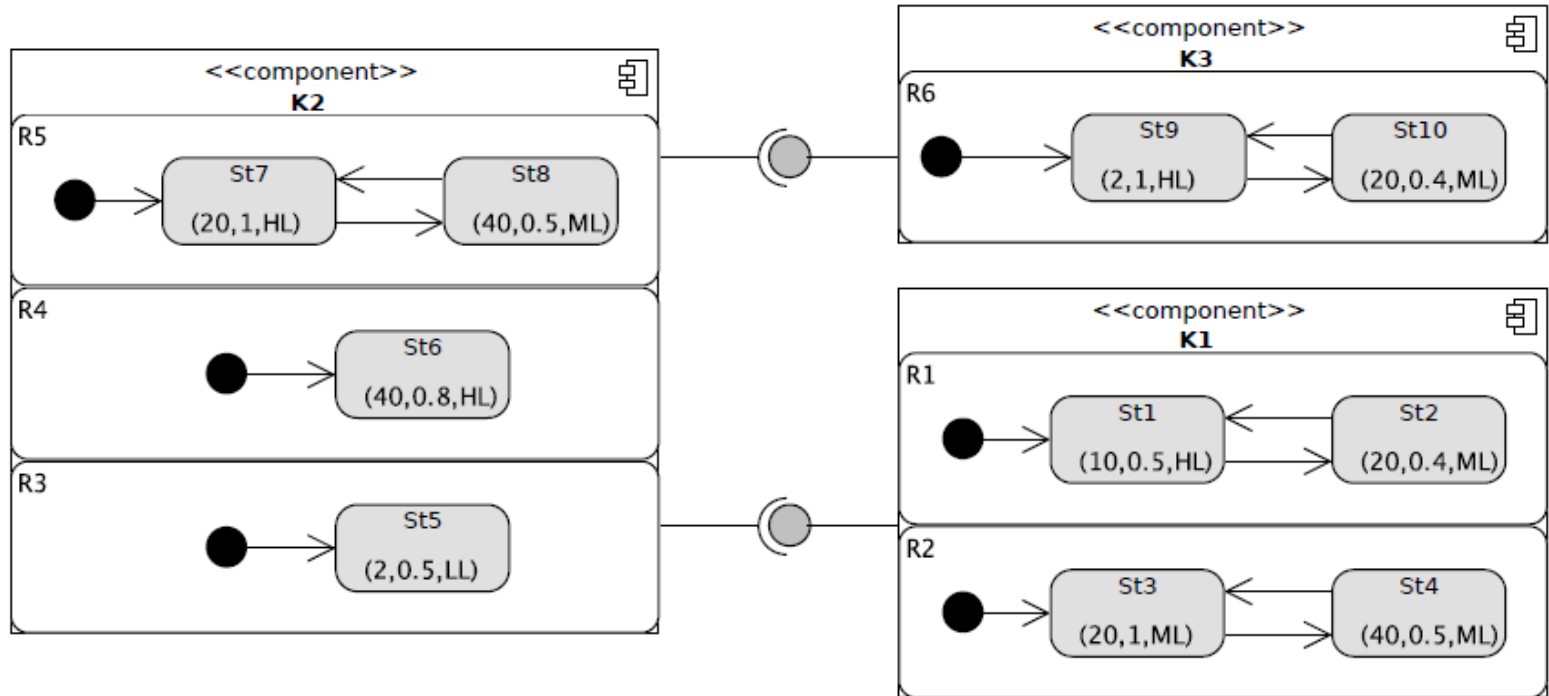
- Threads belonging to **each band are scheduled** by following the **rate monotonic algorithm**.

Threads in the HP band will have greater priority than threads in the MP band, independently of their period.

- **A cyclic executive scheduler is created inside each thread** in order to control the execution of the regions assigned to it.

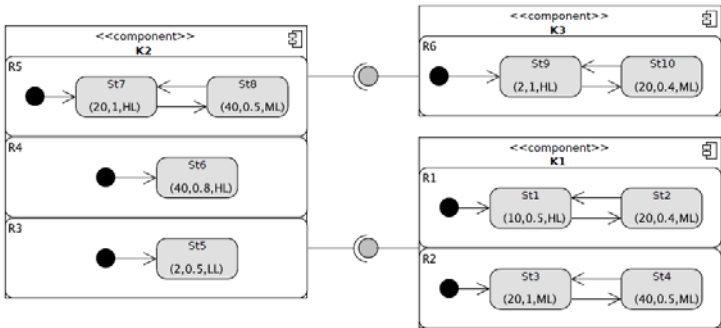
$$H^i = lcm(T_{reg} \in Th^i)$$

# Sample Application. Regions.



Region	Period (ms)	WCET (ms)
R1	10	0.5
R2	20	1
R3	2	0.5
R4	40	0.8
R5	20	1
R6	2	1

# Sample Application. Threads.



Region	Period (ms)	WCET (ms)
R1	10	0.5
R2	20	1
R3	2	0.5
R4	40	0.8
R5	20	1
R6	2	1

**Region to threads assignment. A possible scheme:**

Thread	Region/s	Period (ms)	WCET (ms)	Priority
Th1	R2	20	1	4
Th2	R1, R4, R5	10	2.3	3
Th3	R3	2	0.5	2
Th4	R6	2	1	1

## Scheduling regions inside threads.

- Th2 does need to schedule R1;R4, and R5.
- **Primary cycle**  $H2 = \text{lcm}(TR1; TR4; TR5) = \text{lcm}(10\text{ms}; 40\text{ms}; 20\text{ms}) = 40\text{ms}$ .
- **Secondary cycle** coincides with the thread period,  $Ts2 = 10\text{ms}$ .

Scheduling table will have four secondary cycles of 10ms each:

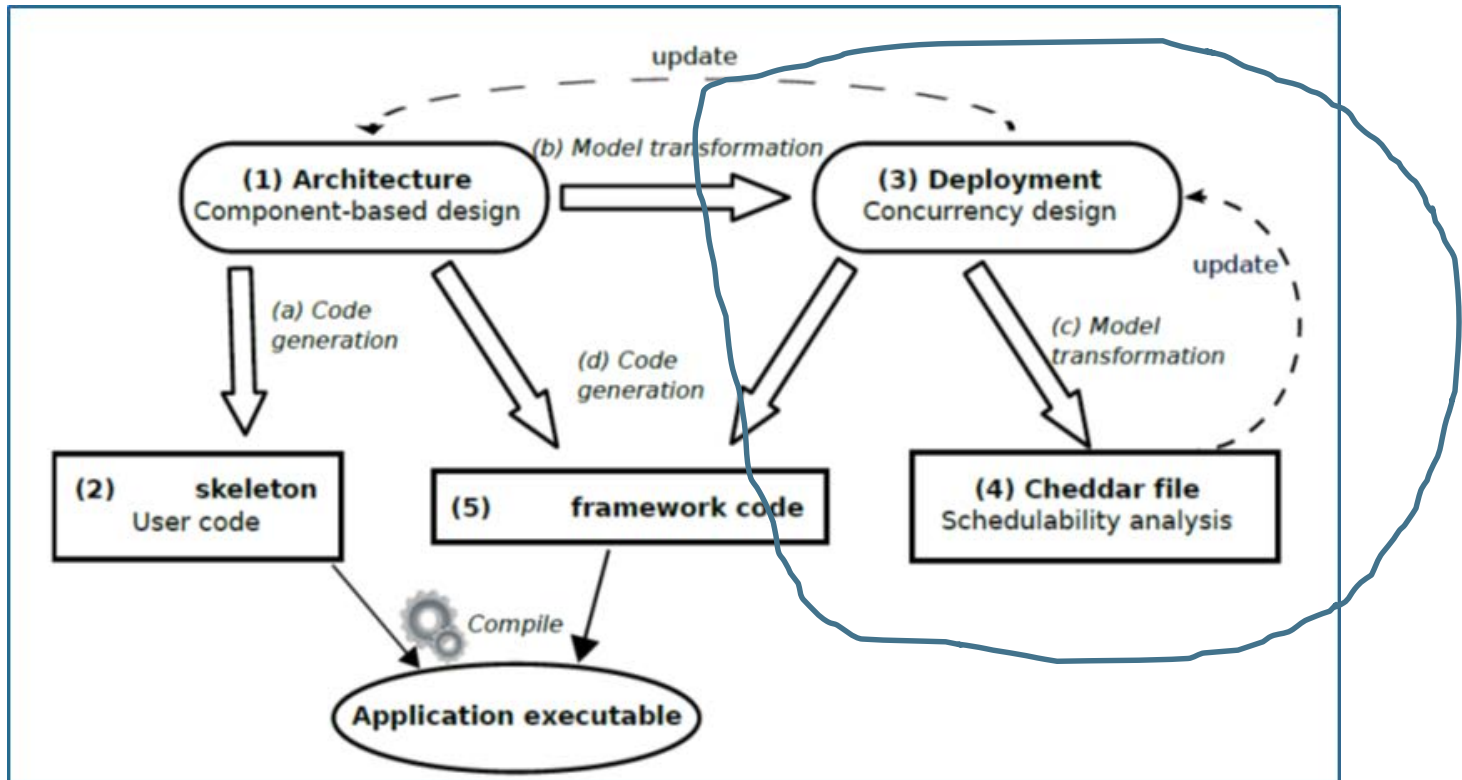
$t = 0\text{ms}$  Executes R1, R4 and R5

$t = 10\text{ms}$  Executes R1

$t = 20\text{ms}$  Executes R1 and R5

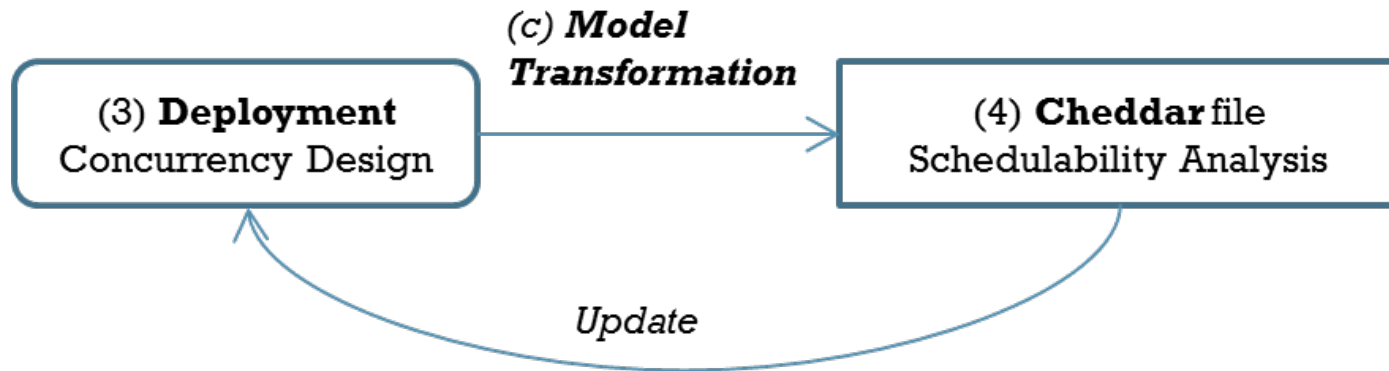
$t = 30\text{ms}$  Executes R1

# Analysis Model (Cheddar).



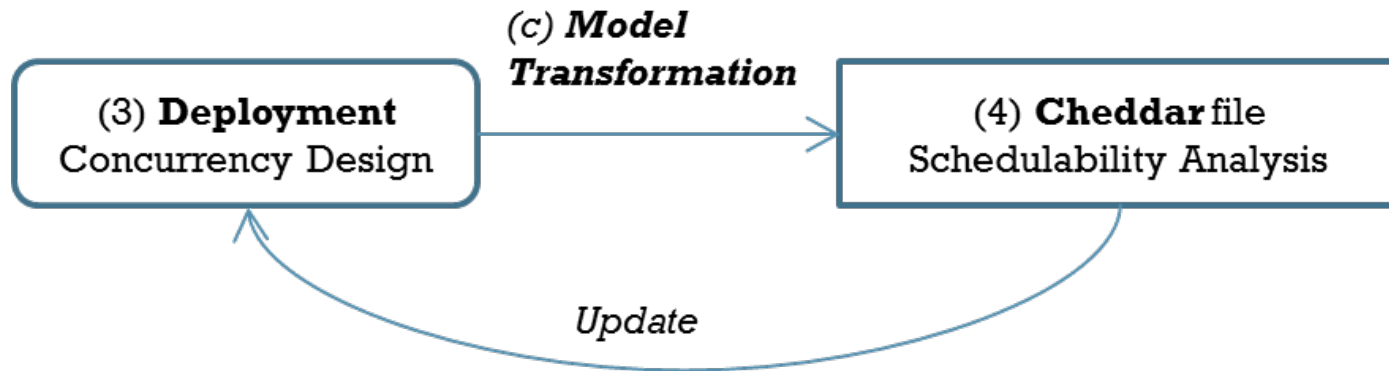
- **Cheddar** is a RT scheduling tool, designed for checking task temporal constraints of a RT system.
- It requires the number of tasks, their timing properties and the number of shared resources of the application.

# Analysis Model (Cheddar).



- **Threads** are directly transformed into **Cheddar tasks**.
- **Shared resources** must be derived from the deployment model. According to the memory structure, **only message buffers** are candidates to be shared among threads.
- Shared resources do not use synchronization primitives, **only mutual exclusion** (**communication** among components is **always asynchronous**).
- Only those buffers that hold messages contained in regions assigned to different threads need to be protected from concurrent access.

# Analysis Model (Cheddar).



- All the needed information can be derived from the architectural and deployment models.
- **If the schedulability fails**, the developer can:
  - Generate new deployment models**, by changing the number of threads and the assignment of regions to threads.
  - Modify the code or the algorithms** used in the activities to faster ones, or by relaxing the timing constraints of the states.
  - Change the components themselves**, and thus the application architecture.



# Sample Application. Deployments.

- The **default deployment model** created by the Fracc Toolchain, defines **one node** with a **single process** hosting just **one thread**.

Deployment 1 (T, WCET)
<b>Thread1 (T=2, WCET=4.8)</b>
Reg1 (10, 0.5)
Reg2 (20, 1.0)
Reg3 (2, 0,5)
Reg4 (40, 0.8)
Reg5 (20, 1.0)
Reg6 (2, 1.0)

Deployment 2
<b>Thread1 (T=10, WCET=1.5)</b>
Reg1 (10, 0.5)
Reg2 (20, 1.0)
<b>Thread2 (T=2, WCET=1)</b>
Reg6 (2, 1.0)
<b>Thread3 (T=2, WCET=2.3)</b>
Reg3 (2, 0,5)
Reg4 (40, 0.8)
Reg5 (20, 1.0)

## Cheddar analysis results:

### *Feasibility test based on the processor utilization factor:*

- Processor utilization factor with deadline is 2.4
- In the pre-emptive case, with RM, cannot prove that the task set is schedulable: processor utilization factor is more than 1.0

### *Feasibility test based on worst case task response time:*

Processor utilization exceeded: cannot compute bound on the response time with this task set.

# Sample Application. Deployments.

## Deployment 3

Thread1 (T=10, WCET=1.5)

Reg1 (10, 0.5)

Reg2 (20, 1.0)

Thread2 (T=20, WCET=1.8)

Reg4 (40, 0.8)

Reg5 (20, 1.0)

Thread3 (T=2, WCET=1.5)

Reg3 (2, 0.5)

Reg6 (2, 1.0)

### Feasibility test based on the processor utilization factor:

- Utilization Factor: 0.99
- 200  $\mu\text{s}$  unused in the base period.
- In the pre-emptive case, with RM, the task set is schedulable.

### Feasibility test based on worst case response time:

- Th2: 19800  $\mu\text{s}$ , Th1: 6000  $\mu\text{s}$ , Th3: 1500  $\mu\text{s}$

## Deployment 4

Thread1 (T=10, WCET=0.5)

Reg1 (10, 0.5)

Thread2 (T=20, WCET=1.0)

Reg2 (20, 1.0)

Thread3 (T=2, WCET=0.5)

Reg3 (2, 0.5)

Thread4 (T=40, WCET=0.8)

Reg4 (40, 0.8)

Thread5 (T=20, WCET=1.0)

Reg5 (20, 1.0)

Thread6 (T=2, WCET=1.0)

Reg6 (2, 1.0)

### Feasibility test based on the processor utilization factor:

- Utilization Factor: 0.92
- 3200  $\mu\text{s}$  unused in the base period.
- In the pre-emptive case, with RM, the task set is schedulable.

### Feasibility test based on worst case response time:

- Th4: 15800  $\mu\text{s}$ , Th2: 10000  $\mu\text{s}$ , Th5: 6000  $\mu\text{s}$
- Th1: 2000  $\mu\text{s}$ , Th3: 1500  $\mu\text{s}$ , Th6: 1000  $\mu\text{s}$

# Overview of WCOMM

The screenshot displays the Eclipse IDE interface for WCOMM development. The main workspace is divided into several panes:

- Project Explorer:** Shows the project structure for 'auvb1', including source files like 'auvb1\_act.wida', 'auvb1\_ifc.wida', and 'basicDatatypes.wida', as well as system libraries and other project-specific files.
- Diagram View (Top):** Displays the 'C\_MissionInterface' diagram. It features a state machine with states 'S.Reporting' and 'A.Reporting', and a stateful component 'A.ProcessingCmds'. A palette on the right lists elements like 'SimpleComponent', 'Port', 'XTA', and 'XTA' transitions.
- Diagram View (Bottom):** Shows the 'auvb1\_app.wccd\_diagram', a component diagram illustrating the relationships between 'C\_MissionInterface', 'C\_MissionSpooler', 'C\_PathPlanner', 'C\_HAL\_AUV', and 'C\_ObstacleAvoider'. It includes interfaces like 'I\_MissionSts', 'I PlanCmd', 'I PathPlan', 'I AuvSts', 'I WayPointReq', and 'I ObsAvoid'.
- Code Editor:** Displays the 'auvb1\_ifc.wida' file, containing the definition of primitive data types and interfaces:

```
// definition of primitive data types
import "basicDatatypes.wida";
import "userDatatypes.wida";
package auvb1.interfaces;

interface I_PlanCmd {
    message setPlan (int32 numSteps,
                    message runPlan ();
    message abort ();
    message pause ();
    message resume ();
}

interface I_MissionSts {
    message nicSts (planSts sts);
}

interface I_PathPlan{
    message pathReq (pose start, pose
                    message path (int32 numWaypoints,
}

interface I_ObsAvoid {
    message enable();
    message disable();
    message obvSts (int16 sts);
}

interface I_AuvSts{
    message auvSts (int16 sts);
}

interface I_WaypointReq{
    message waypointReq (pose waypoint
}

interface I_VelocityReq{
    message velocityReq (vel1 velocity
}
```

# Messages, datatypes and activities

The image displays the Eclipse IDE environment with two main diagrams and a code editor.

**Top Diagram: C\_MissionInterface**  
This diagram shows the internal structure of the mission interface. It features a central stateful component labeled 'A.ProcessingOnes' with four internal states represented by small squares. This component is connected via message links to external components: 'I.Planning' on the left, 'I.PathPlan' on the right, and 'I.ObsAvoid' at the bottom right. A palette on the right side of the diagram lists various modeling elements like 'Port', 'XTA', and 'Transition'.

**Bottom Diagram: \*auvbl\_app.wcccd\_diagram**  
This diagram illustrates the system architecture through component dependencies. At the top is 'C\_MissionInterface'. Below it are 'I\_MissionSts' and 'I\_PlanCmd'. 'C\_MissionInterface' depends on both. 'I\_MissionSts' depends on 'C\_MissionSpooler'. 'I\_PlanCmd' depends on 'C\_PathPlanner'. 'C\_MissionSpooler' depends on 'I\_AuvSts' and 'I\_WayPointReq'. 'C\_PathPlanner' depends on 'I\_PathPlan' and 'I\_ObsAvoid'. 'I\_AuvSts' depends on 'C\_HAL\_AUV'. 'I\_WayPointReq' depends on 'C\_PathPlanner'. 'I\_ObsAvoid' depends on 'C\_PathPlanner' and 'C\_ObstacleAvoider'. A palette on the right lists modeling elements like 'Component', 'Port', and 'Links'.

**Code Editor: \*auvbl\_#c.wida**  
The code editor shows the definition of data types and interfaces. A blue arrow points to the start of the code. The code includes imports for 'BasicDataTypes', 'UserDataTypes', and 'wida'. It defines several interfaces with message methods:

```
// definition of ... ive data types
import "BasicDataTypes.wida";
import "UserDataTypes.wida";
package auvbl...

interface I_PlanCmd {
    message setPlan (int32 numSteps, ...);
    message runPlan ();
    message abort ();
    message pause ();
    message resume ();
}

interface I_MissionSts {
    message nisSts (planSts sts);
}

interface I_PathPlan {
    message pathReq (pose start, pose ...);
    message path (int32 numWaypoints, ...);
}

interface I_ObsAvoid {
    message enable();
    message disable();
    message obsAvSts (int16 sts);
}

interface I_AuvSts {
    message auvSts (int16 sts);
}

interface I_WaypointReq {
    message waypointReq (pose waypoint ...);
}

interface I_VelocityReq {
    message velocityReq (vel velocity ...);
}
```

# Simple Components (plus finite-state machine)

The image displays the Eclipse IDE interface with two main windows. The top window, titled "C\_MissionInterface.wscd\_diagram", shows a state machine diagram for the "C\_MissionInterface" component. The diagram includes states like "S.Reporting" and "A.ProcessingCmds", and transitions between them. A palette on the right lists elements like "SimpleComponent", "Port", "XTA", "Transition", "Initial Pseudostate", "Final State", "Bindings", "Event Link", "Message Link", "Clock Link", and "MsgEventLink". A large blue arrow points from the top of the diagram towards the state machine elements.

The bottom window, titled "auvb1\_app.wscd\_diagram", shows a component dependency graph. It illustrates the relationships between various components and interfaces, including "C\_MissionInterface", "I\_MissionSts", "C\_MissionSpooler", "I\_AuvSts", "I\_WayPointReq", "I\_PlanCmd", "I\_PathPlan", "C\_PathPlanner", "I\_ObsAvoid", "C\_OBS\_AUV", "I\_ObsAvoid", and "C\_ObstacleAvoider".

On the right side of the IDE, there is a code editor showing the definition of primitive data types and interfaces:

```

// definition of primitive data types
import "basicDatatypes.wide";
import "userDatatypes.wide";
package auvb1.interfaces;

interface I_PlanCmd {
    message setPlan (int32 numSteps,
                    message runPlan ();
                    message abort ();
                    message pause ();
                    message resume ();
}

interface I_MissionSts {
    message niaSts (planSts sts);
}

interface I_PathPlan{
    message pathReq (pose start, pose
                    message path (int32 numWaypoints,
}

interface I_ObsAvoid {
    message enable();
    message disable();
    message obsAvSts (int16 sts);
}

interface I_AuvSts{
    message auvSts (int16 sts);
}

interface I_WaypointReq{
    message waypointReq (pose waypoint
}

interface I_VelocityReq{
    message velocityReq (vel velocity
}

```

The screenshot displays the Eclipse IDE with a UML Component Diagram for an AUV system. The diagram is divided into two main views:

- Top View (C\_MissionInterface):** Shows a detailed view of the `C_MissionInterface` component. It includes a state machine diagram with states like `A_Spawning` and `A_ProcessingCmds`. It also shows connections to other components like `C_PathPlanner` and `C_ObstacleAvoider`.
- Bottom View (\*auv1\_app.wccd\_diagram):** Shows a high-level component diagram. The components are represented as yellow boxes:
  - `C_MissionInterface` (top)
  - `C_MissionSpooler` (middle)
  - `C_PathPlanner` (right)
  - `C_HAL_AUV` (bottom left)
  - `C_ObstacleAvoider` (bottom right)
 Interfaces are shown as white boxes with arrows pointing to the components that implement them:
  - `I_MissionSts` (implements `C_MissionInterface` and `C_MissionSpooler`)
  - `I_PlanCmd` (implements `C_MissionInterface`)
  - `I_PathPlan` (implements `C_PathPlanner`)
  - `I_AuvSts` (implements `C_HAL_AUV`)
  - `I_WayPointReq` (implements `C_PathPlanner` and `C_ObstacleAvoider`)
  - `I_ObsAvoid` (implements `C_ObstacleAvoider`)

On the right side, there is a code editor showing the definition of primitive data types and interfaces:

```

// definition of primitive data types
import "basicDatatypes.wide";
import "userDatatypes.wide";
package auv1.interfaces;

interface I_PlanCmd {
    message setPlan (int32 numSteps,
                    message runPlan ();
    message abort ();
    message pause ();
    message resume ();
}

interface I_MissionSts {
    message niaSts (planSts sts);
}

interface I_PathPlan{
    message pathReq (pose start, pose
    message path (int32 numWaypoints,
}

interface I_ObsAvoid {
    message enable();
    message disable();
    message obsAvSts (int16 sts);
}

interface I_AuvSts{
    message auvSts (int16 sts);
}

interface I_WaypointReq{
    message waypointReq (pose waypoint
}

interface I_VelocityReq{
    message velocityReq (vel velocity
}
  
```

A blue arrow points to the `C_ObstacleAvoider` component in the bottom diagram.

## Complex Components and Application

# Conclusions and further work

The presented approach provides:

- Control over the concurrency characteristics of the application and
- Schedulability analysis of execution model.

These objectives have been achieved by

- Defining a **component model** that includes **structure** and **behaviour**;
- Describing **temporal requirements** at the **architectural level**;
- **Decoupling the structural elements** from the **behavioural** and the **algorithmic ones**;
- Defining a clear and consistent **association between the elements of the system** and **execution models** through a **deployment model**.

The approach is supported by a model-driven toolchain developed in Eclipse (C-Forge).

# Conclusions and further work

The **deployment model** has proven to be essential in the approach, since

- it **separates application architecture from its deployment** in terms of nodes, processes and threads, enabling
- the **rapid testing** of different deployment scenarios.
- It does not enforce a rigid association between components and processes/threads, but it **can be easily configured** thanks to the deployment model.

Components are not forced to use a communication software for message exchange in all scenarios, but only on those where the application is distributed in more than one node.



# Conclusions and further work

Regarding future works,

- we are currently **enhancing the deployment model** for:
  - Supporting **multi-core systems**, and end-to-end transactions specification.
  - **Automatically generating and testing** different deployments, in order to find an optimum one.
- We are also interested in generating a **less pessimistic analysis file**.
  - A more exhaustive analysis of the state-machines will enable us to make less pessimistic analysis.

