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
etsit escuela técnica superior de ingeniería de telecomunicación

Contents

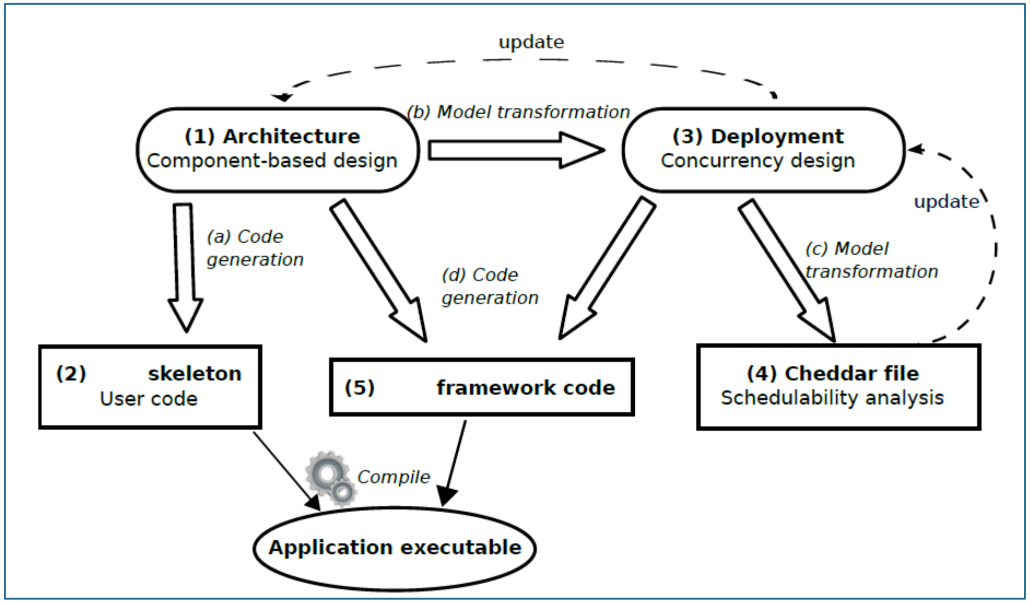
- Technological context: CBSD, MDSD, Cforge Tool Chain.
- General Approach: Solution drivers, the WCOMM component metamodel.
- Task Based-Concurrency Scheme.
- Execution Model.
- Example.
- Deployment and Analysis.
- Conclusions and further work.

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.

MDSD
to the rescue …

Execution model concepts.

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

↑ Very suitable for application construction.

↓ Hinder performance analysis schedulability.

How to map the concepts?

↑ 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 an 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.

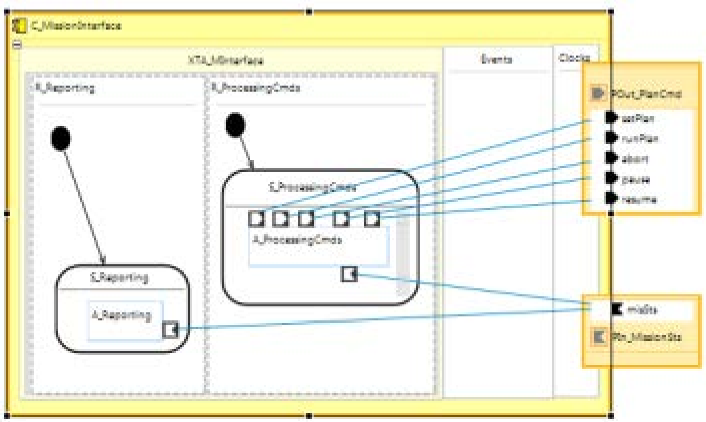
- Create a cyclic executive inside each thread in order to schedule region execution.

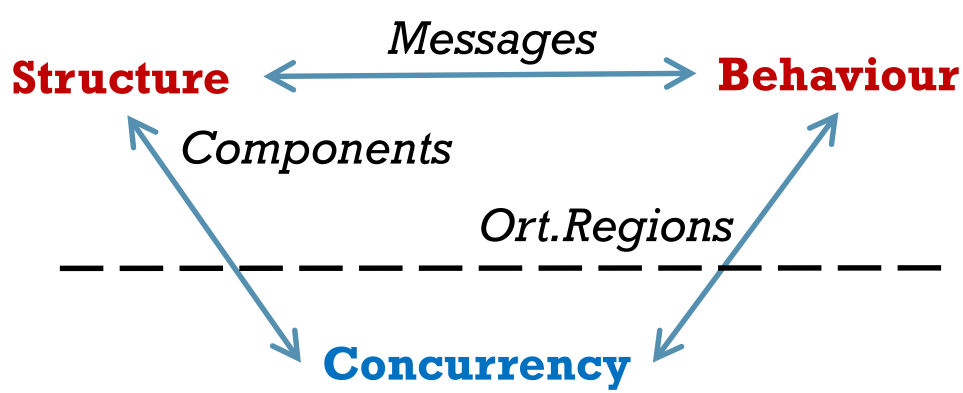
General Approach. The WCOMM Component

- **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

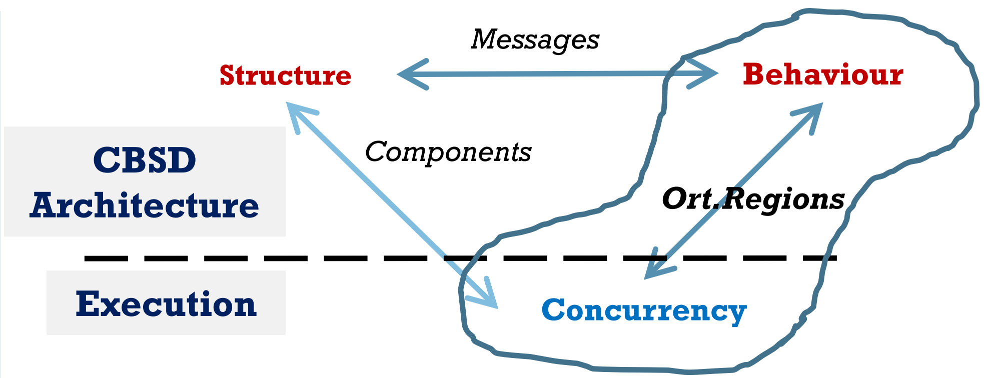
Ports are **flow ports**:

- **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**

App = { K_1, …, K_N }

Components: Set of **Orth. Regions**

K_i = { R_{i1}, …, R_{ir} }

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

R_{ij} = { St_{ij1},..St_{ijs}}

States: Execute **ONE Activity** or none

Activities model **sequential tasks**

Orthogonal region timing properties.
$T^i_{reg} = gcd(T_{act} \in R^i)$
$WCET^i_{reg} = max(WCET_{act} \in R^i)$
$CL^i_{reg} = max(CL_{act} \in R^i)$

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.
$T^i_{reg} = gcd(T_{act} \in R^i)$
$WCET^i_{reg} = max(WCET_{act} \in R^i)$
$CL^i_{reg} = max(CL_{act} \in R^i)$

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

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

Nodes contain **processes.**	$N_i = \{ Pr_{i1}, \ldots, Pr_{ir} \}$
Components are assigned to **proceses**	$Pr_{ij} \ \{ K_{ijk} \}$

Processes contain **threads.**	$Pr_{ij} \ \{ Th_{ijt} \}$

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

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

→

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

↑

Threads properties can be derived from the assigned regions.

↑

Threads timing properties.
$T^i_{th} = gcd(T_{reg} \in Th^i)$
$WCET^i_{th} = \sum(WCET^i_{reg} \in Th^i)$
$PB^i_{th} = max(CL_{reg} \in Th^i)$

Execution Model. Mixed Criticality.

Orthogonal region timing properties.
$T^i_{reg} = gcd(T_{act} \in R^i)$
$WCET^i_{reg} = max(WCET_{act} \in R^i)$
$CL^i_{reg} = max(CL_{act} \in R^i)$

Threads timing properties.
$T^i_{th} = gcd(T_{reg} \in Th^i)$
$WCET^i_{th} = \sum(WCET^i_{reg} \in Th^i)$
$PB^i_{th} = max(CL_{reg} \in Th^i)$

Criticality Levels

HL > ML > LL

Priority Bands

HP > MP > LP

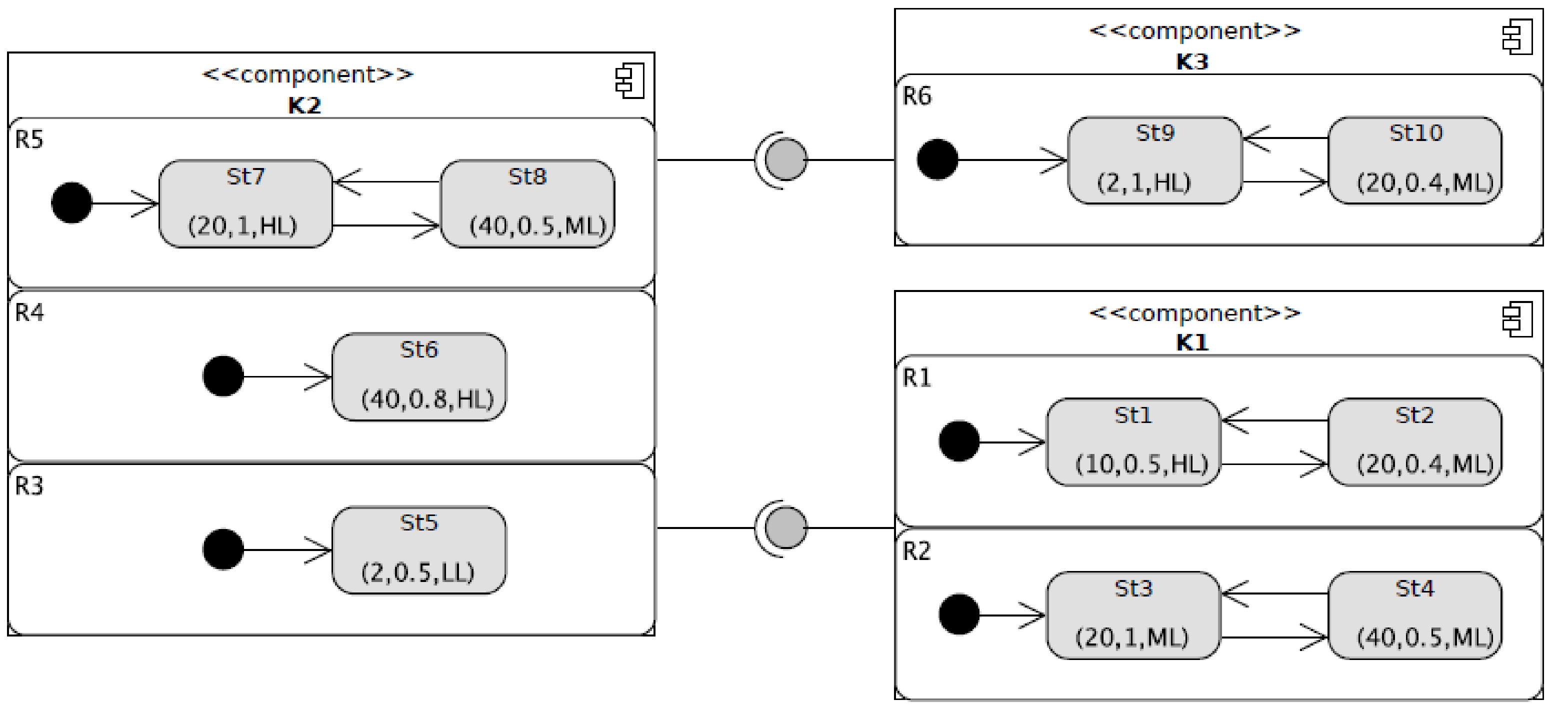
- Threads belonging to **each band are scheduled** by following the **rate monotonic algorithm**.

  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.

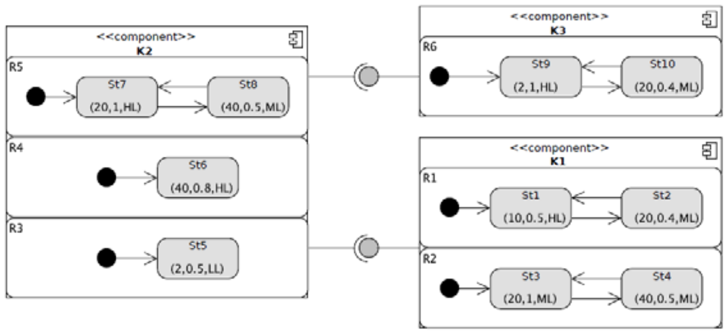
$$H^i = lcm(T_{reg} \in Th^i)$$

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 = lcm(TR1; TR4; TR5) = lcm(10ms; 40ms; 20ms) = 40ms.
- **Secondary cycle** coincides with the thread period, Ts2 = 10ms.

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

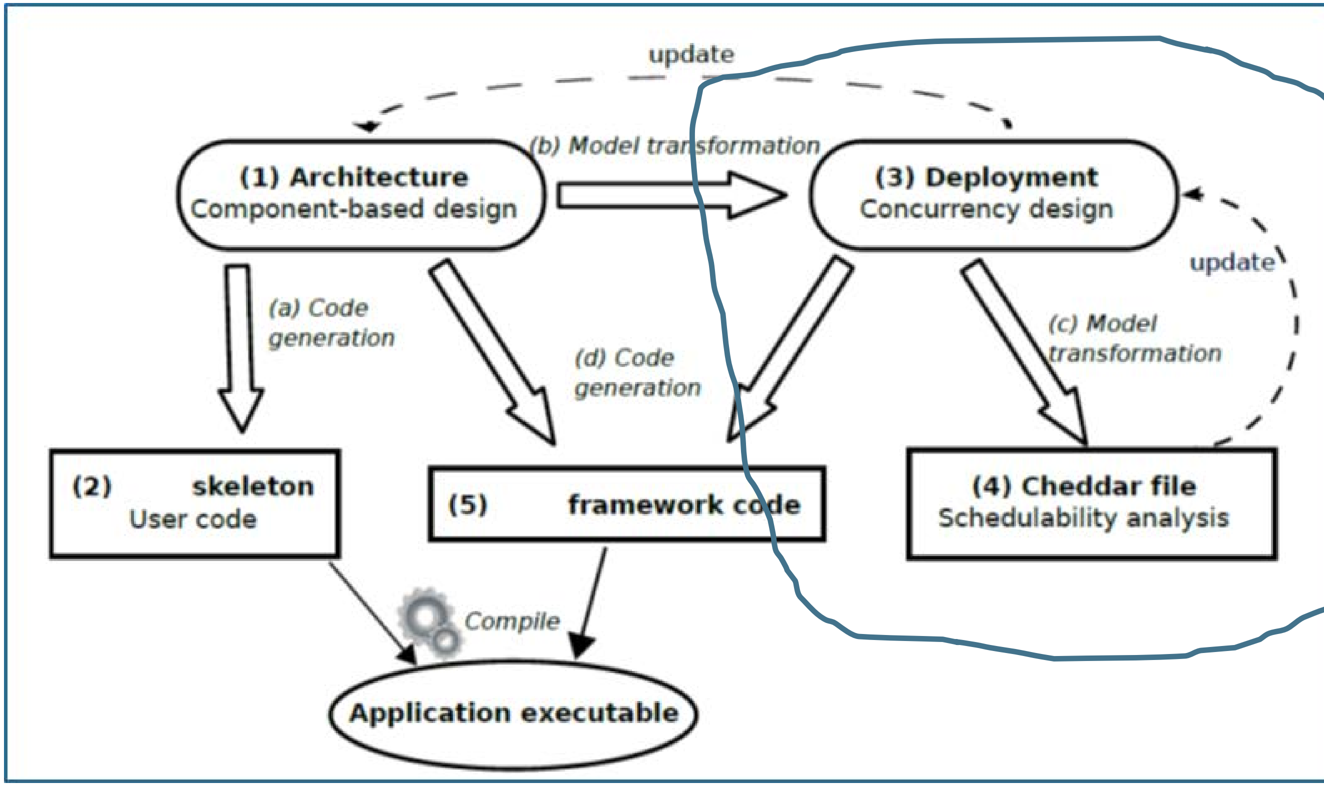
t = 0ms Executes R1, R4 and R5

t = 10ms Executes R1

t = 20ms Executes R1 and R5

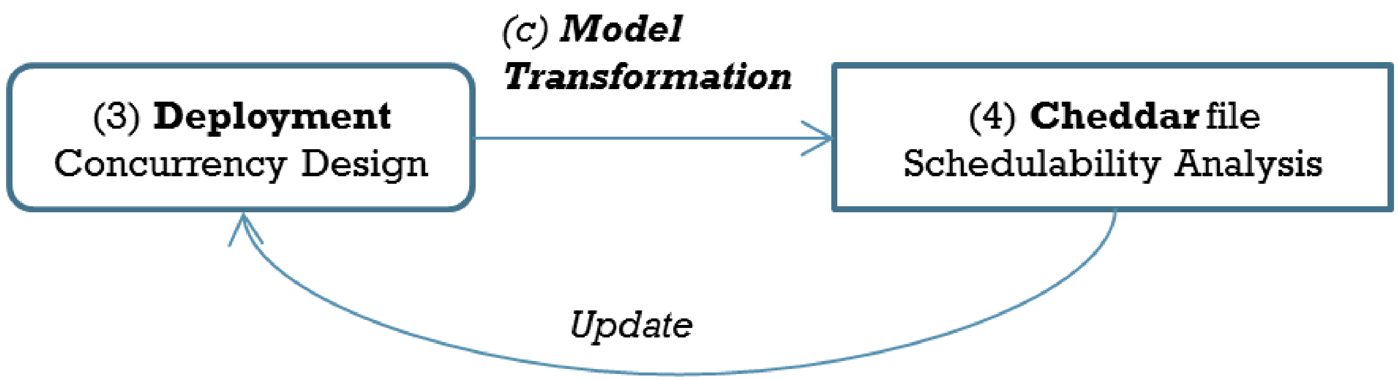
t = 30ms Executes R1

Analysis Model (Cheddar).



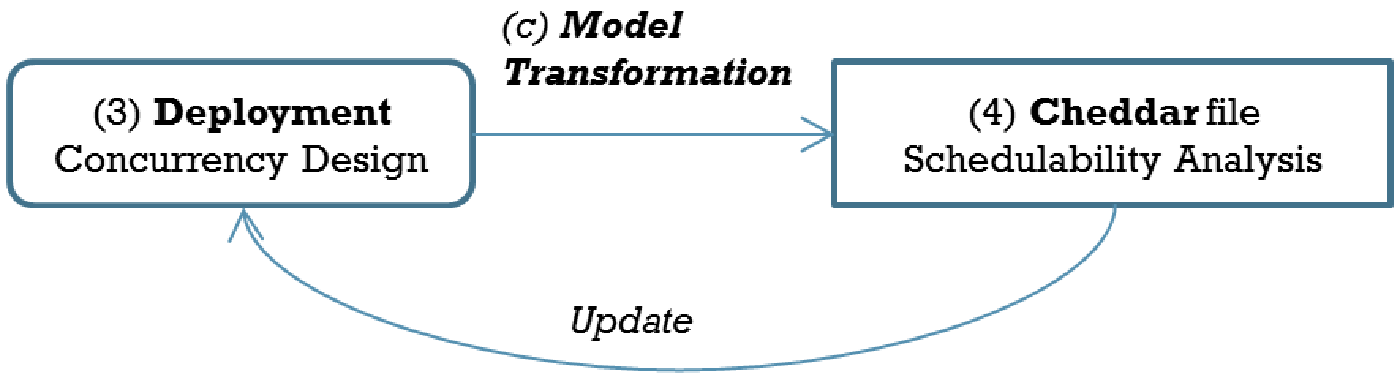
- **Cheddar** is a RT scheduling tool, designed for checking task temporal constraints of a RT system.
- It requires the number of tasks, their timing properties and the number of shared resources of the application.

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)

```
Thread1 (T=2, WCET=4.8)
        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

```
Thread1 (T=10, WCET=1.5)
        Reg1 (10, 0.5)
        Reg2 (20, 1.0)
Thread2 (T=2, WCET=1)
        Reg6 (2, 1.0)
Thread3 (T=2, WCET=2.3)
        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 µs 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 µs, Th1: 6000 µs, Th3: 1500 µs

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)
```

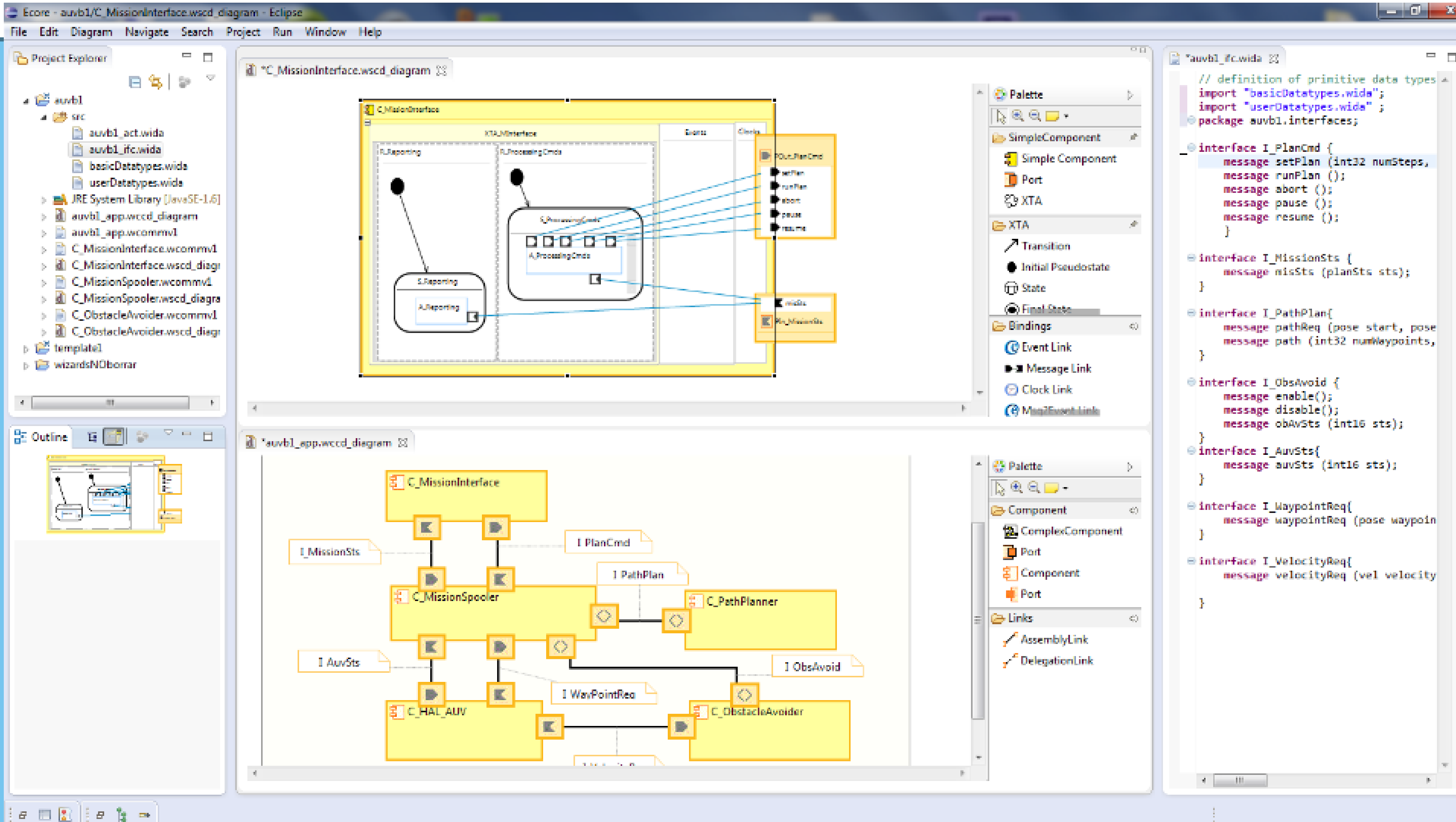
Feasibility test based on the processor utilization factor:

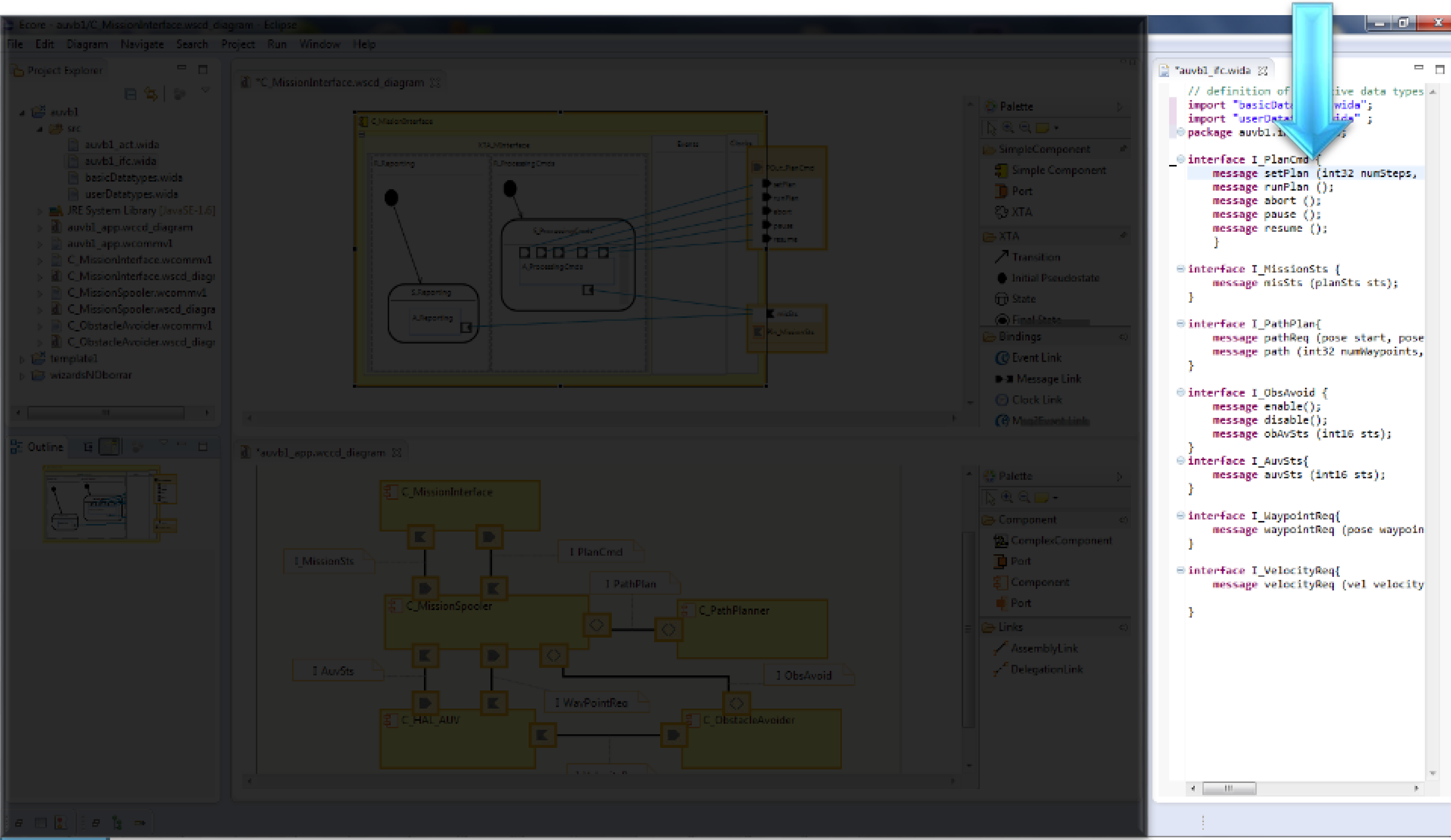
- Utilization Factor: 0.92
- 3200 µs unused in the base period.
- In the pre-emptive case, with RM, the task set is schedulable.

Feasibility test based on worst case response time:

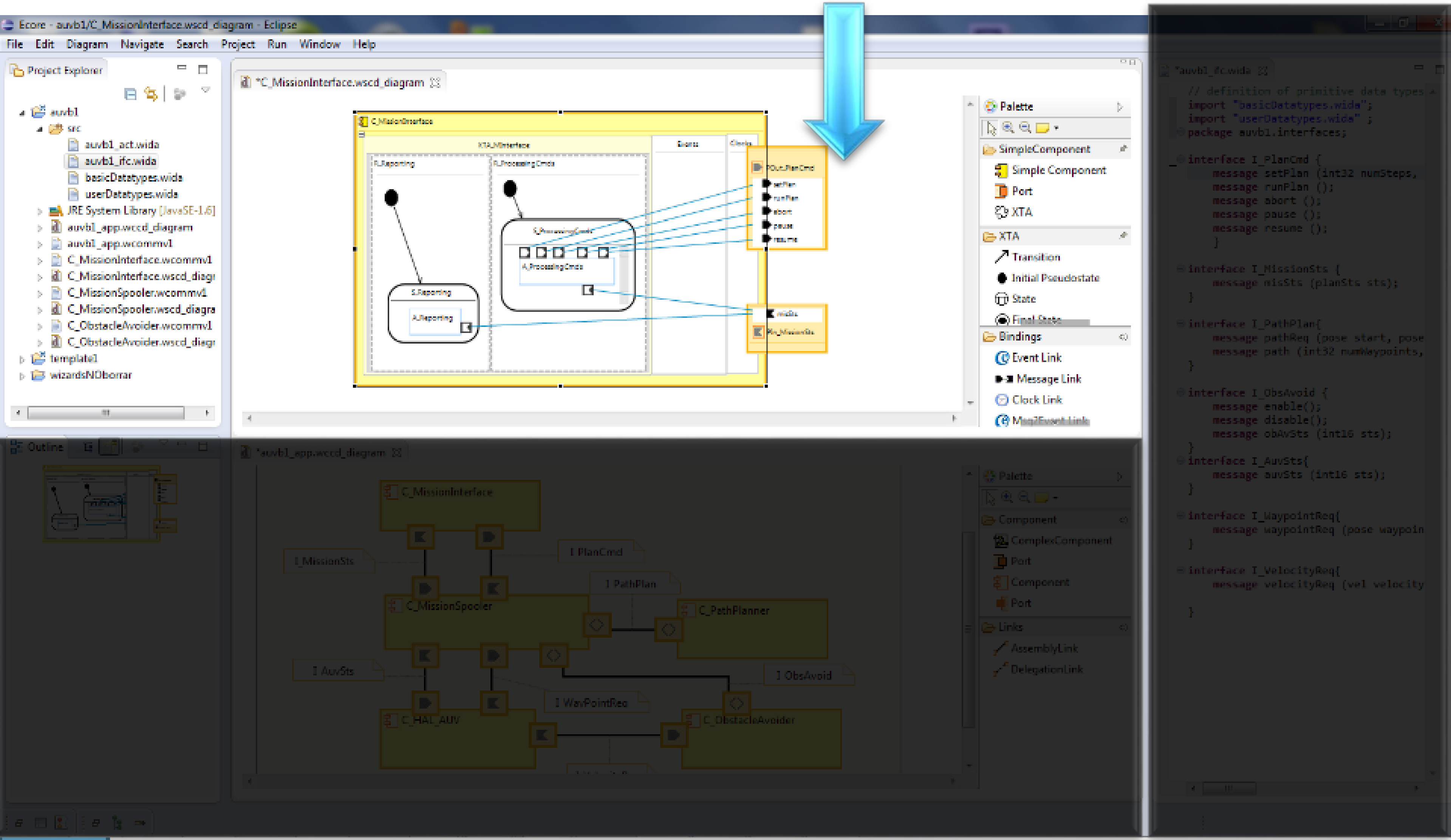
- Th4: 15800 µs, Th2: 10000 µs, Th5:6000 µs
- Th1: 2000 µs, Th3: 1500 µs, Th6:1000 µs

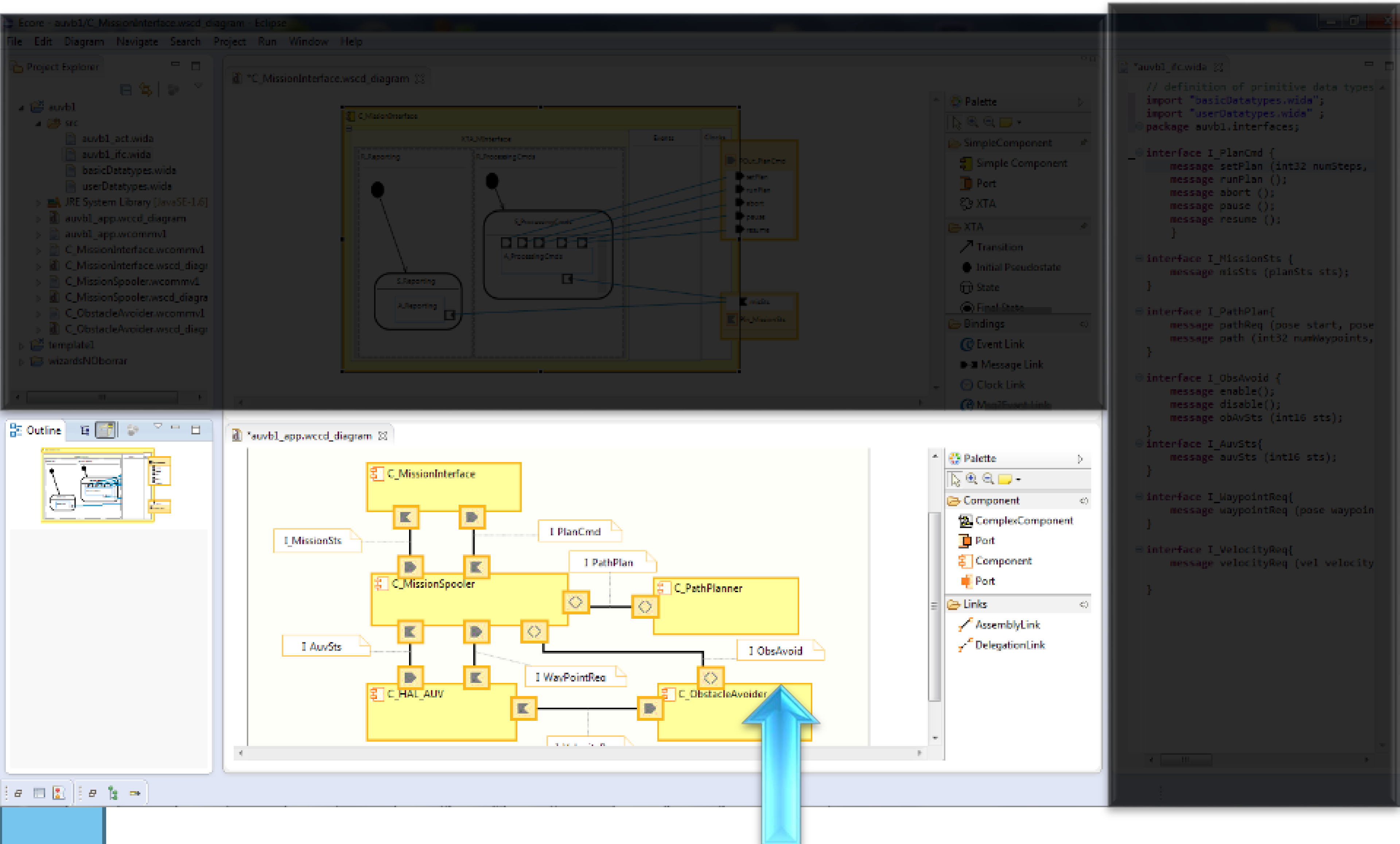
Overview of WCOMM




*auvb1_ifc.wida
import "basicDat
import "userData
package auvb1.i
interface I_PlanCmd {
message setPlan (int32 numSteps,
message runPlan ();
message abort ();
message pause ();
message resume ();
}
interface I_MissionSts {
message misSts (planSts sts);
}
interface I_PathPlan{
message pathReq (pose start, pose
message path (int32 numWaypoints,
}
interface I_ObsAvoid {
message enable();
message disable();
message obAvSts (int16 sts);
}
interface I_AuvSts{
message auvSts (int16 sts);
}
interface I_WaypointReq{
message waypointReq (pose waypoin
}
interface I_VelocityReq{
message velocityReq (vel velocity
}
C_MissionInterface
I_MissionSts
I_PlanCmd
I_PathPlan
C_MissionSpooler
C_PathPlanner
I_AuvSts
I_ObsAvoid
I_WayPointReq
C_HAL_AUV
C_ObstacleAvoider

Simple Components (plus finite-state machine)





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.