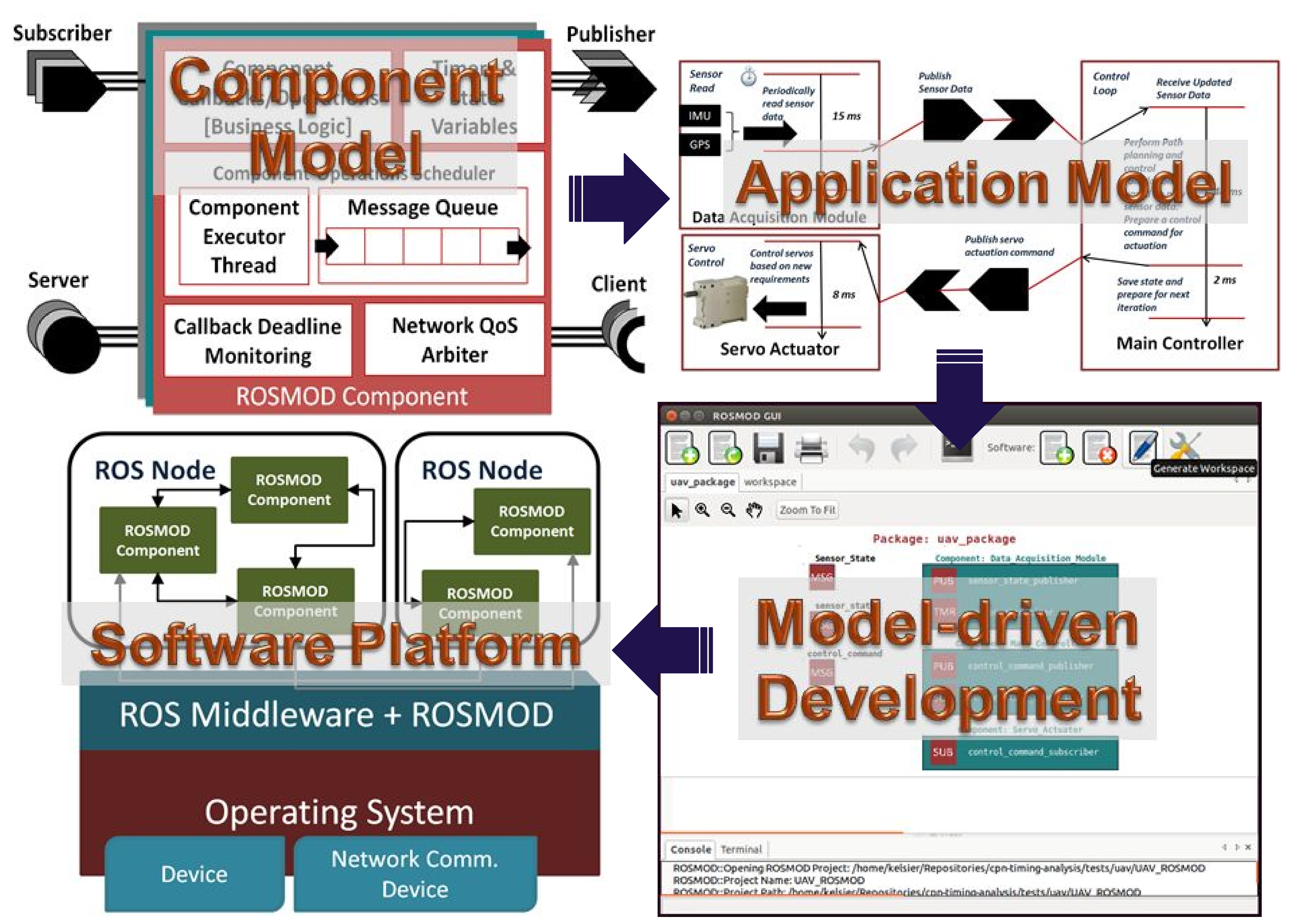
ROSMOD: A Toolsuite for Modeling, Analyzing, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS

Pranav Srinivas Kumar, William Emfinger, and Gabor Karsai

Institute for Software-Integrated Systems

Vanderbilt University

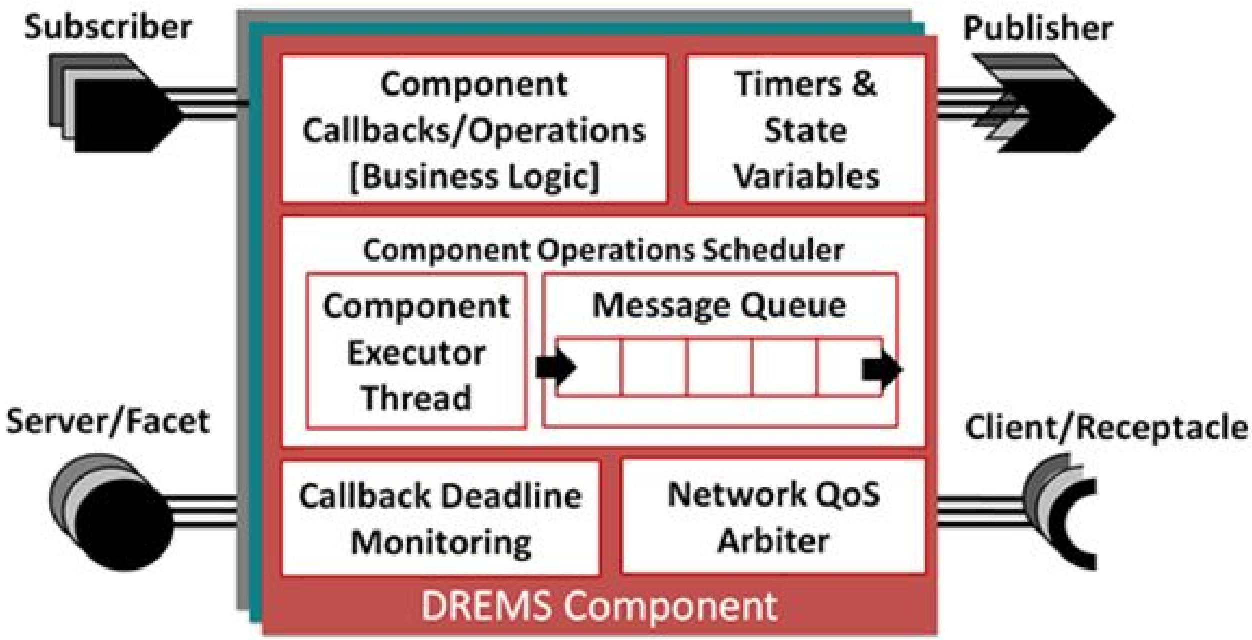
This work was supported by DARPA under contract NNA11AB14C and USAF/AFRL under Cooperative Agreement FA8750-13-2-0050, and by the National Science Foundation (CNS-1035655). The activities of the 2014-15 Vanderbilt Aerospace Club were sponsored by the Department of Mechanical Engineering and the Boeing company. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, USAF/AFRL, NSF, or the Boeing Company.



Subscriber
Publisher
Component Model
Component Callbacks/Operations [Business Logic]
Timers & State Variables
Component Operations Scheduler
Component Executor Thread
Message Queue
Callback Deadline Monitoring
Network QoS Arbiter
ROSMOD Component
Server
Client
Application Model
Sensor Read
IMU
GPS
Periodically read sensor data
15 ms
Data Acquisition Module
Publish Sensor Data
Control Loop
Receive Updated Sensor Data
Perform Path planning and control
Prepare a control command for actuation
Save state and prepare for next iteration
2 ms
Main Controller
Publish servo actuation command
Servo Control
Control servos based on new requirements
8 ms
Servo Actuator
ROS Node
ROSMOD Component
Software Platform
ROS Middleware + ROSMOD
Operating System
Device
Network Comm. Device
ROSMOD GUI
Software:
Generate Workspace
uav_package
workspace
Zoom To Fit
Package: uav_package
Sensor_State
Component: Data Acquisition Module
MSG
PUB
sensor_state_publisher
TMR
control_command
control_command_publisher
Component: Servo_Actuator
SUB
control_command_subscriber
Model-driven Development
Console
Terminal
ROSMOD::Opening ROSMOD Project: /home/kelsier/Repositories/cpn-timing-analysis/tests/uav/UAV_ROSMOD
ROSMOD::Project Name: UAV_ROSMOD
ROSMOD::Project Path: /home/kelsier/Repositories/cpn-timing-analysis/tests/uav/UAV_ROSMOD

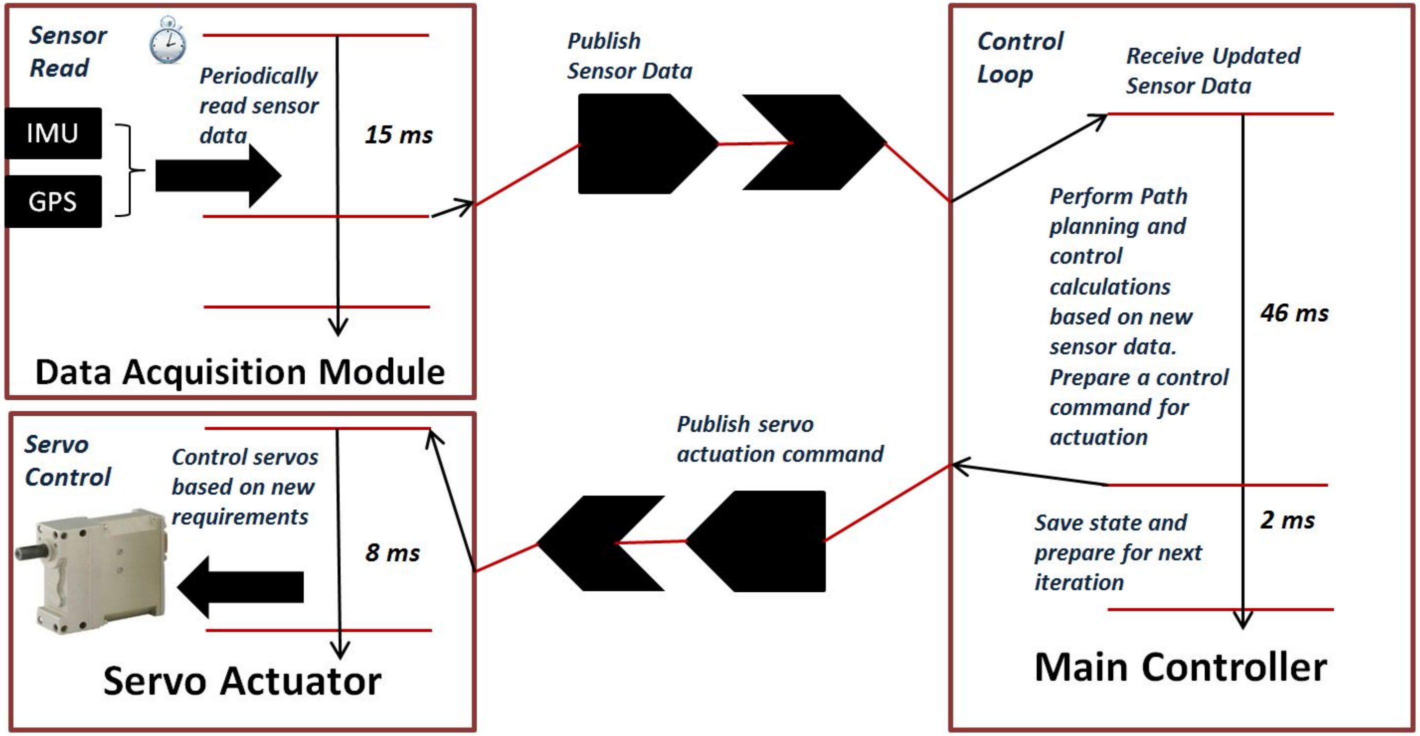
ROSMOD Component

- Each Component has a Message Queue
- Each Component exposes *operations* through port interfaces.
- Message Queue receives operation requests from other components
- Component Operation Scheduler schedules one request at a time from the queue - FIFO, PFIFO or EDF scheduling scheme
- Operation execution is single-threaded (i.e. non-preemptive) per component
- Single threaded operation execution helps avoid synchronization primitives and locking mechanisms in application code

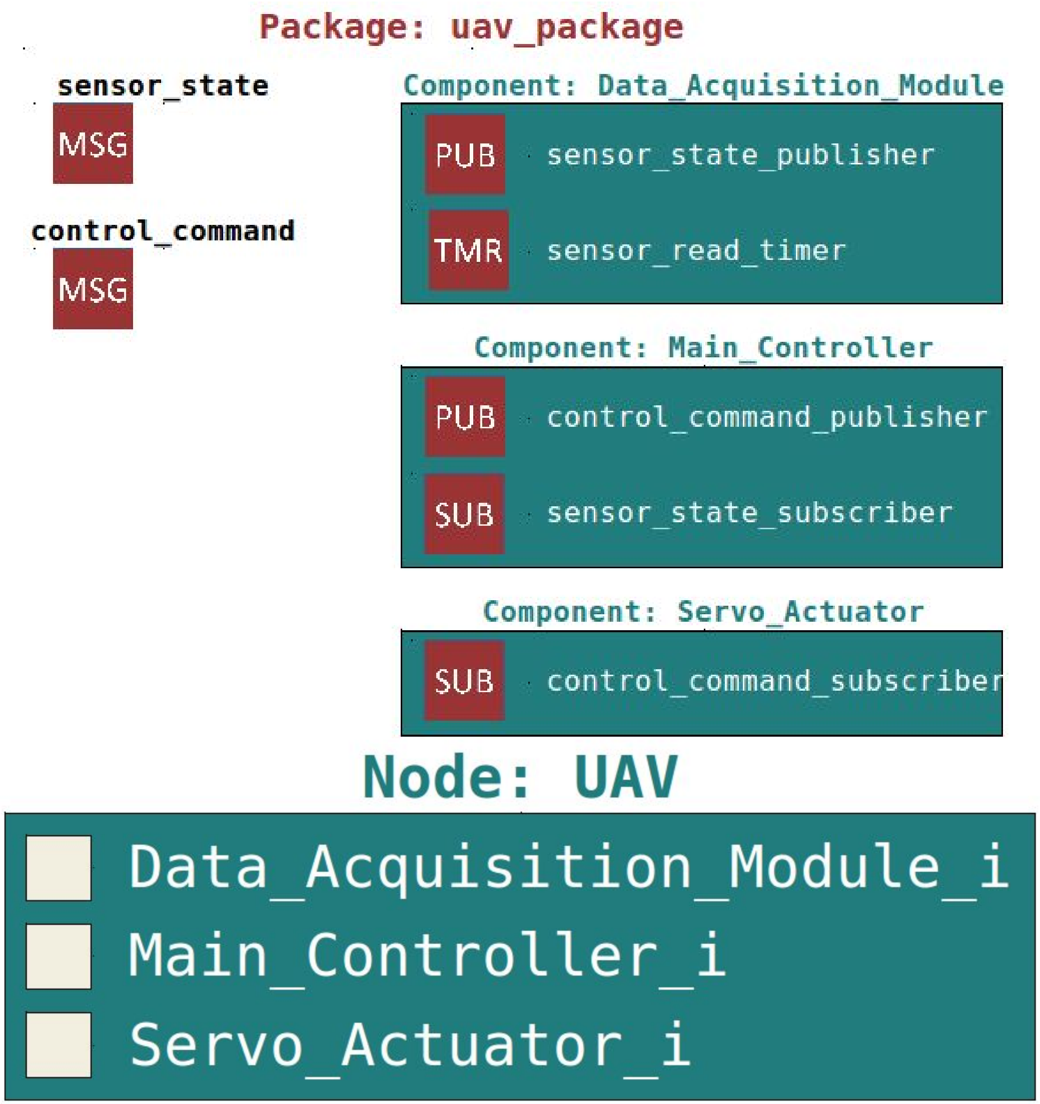


Application Model - Component Assembly Design

- Simple UAV Application
- Periodic timer triggers the Data Acquisition Module to read sensor data from a variety of onboard sensor devices e.g. IMU, GPS
- Sensor Data is packaged into a ROS *msg* and published
- The Main Controller receives this *msg* and performs path planning calculations. The Control Loop operation prepares a control command to actuate the UAV servos accordingly
- The Servo Actuator component receives this control command and controls the servo motors

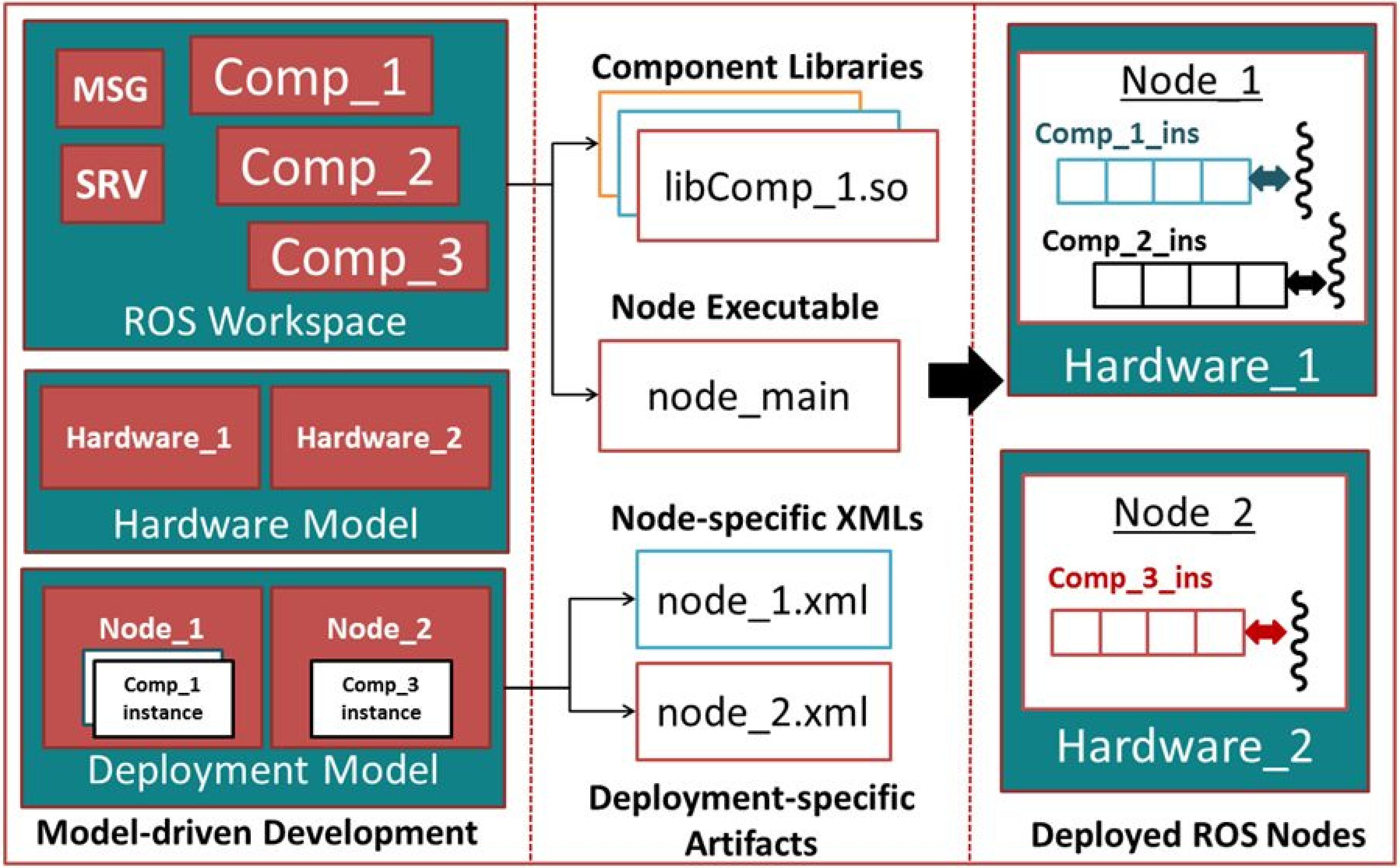


ROSMOD Model-Driven Development



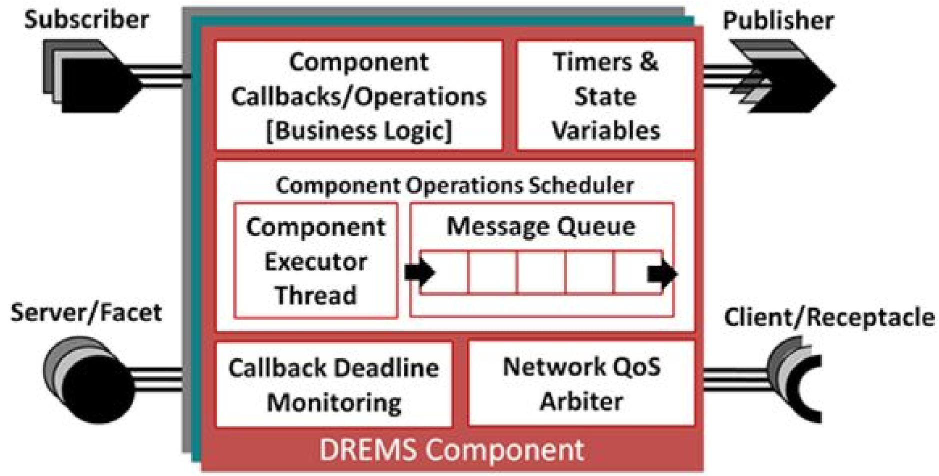

```
.
└── src
    ├── node
    │   ├── CMakeLists.txt
    │   ├── include
    │   │   └── node
    │   │       ├── Component.hpp
    │   │       ├── Logger.hpp
    │   │       ├── rapidxml.hpp
    │   │       ├── rapidxml_utils.hpp
    │   │       └── xmlParser.hpp
    │   ├── package.xml
    │   └── src
    │       └── node
    │           ├── Component.cpp
    │           └── node_main.cpp
    └── uav_package
        ├── CMakeLists.txt
        ├── include
        │   └── uav_package
        │       ├── Data_Acquisition_Module.hpp
        │       ├── Main_Controller.hpp
        │       └── Servo_Actuator.hpp
        ├── msg
        │   ├── control_command.msg
        │   └── sensor_state.msg
        ├── package.xml
        └── src
            └── uav_package
                ├── Data_Acquisition_Module.cpp
                ├── Main_Controller.cpp
                └── Servo_Actuator.cpp
```

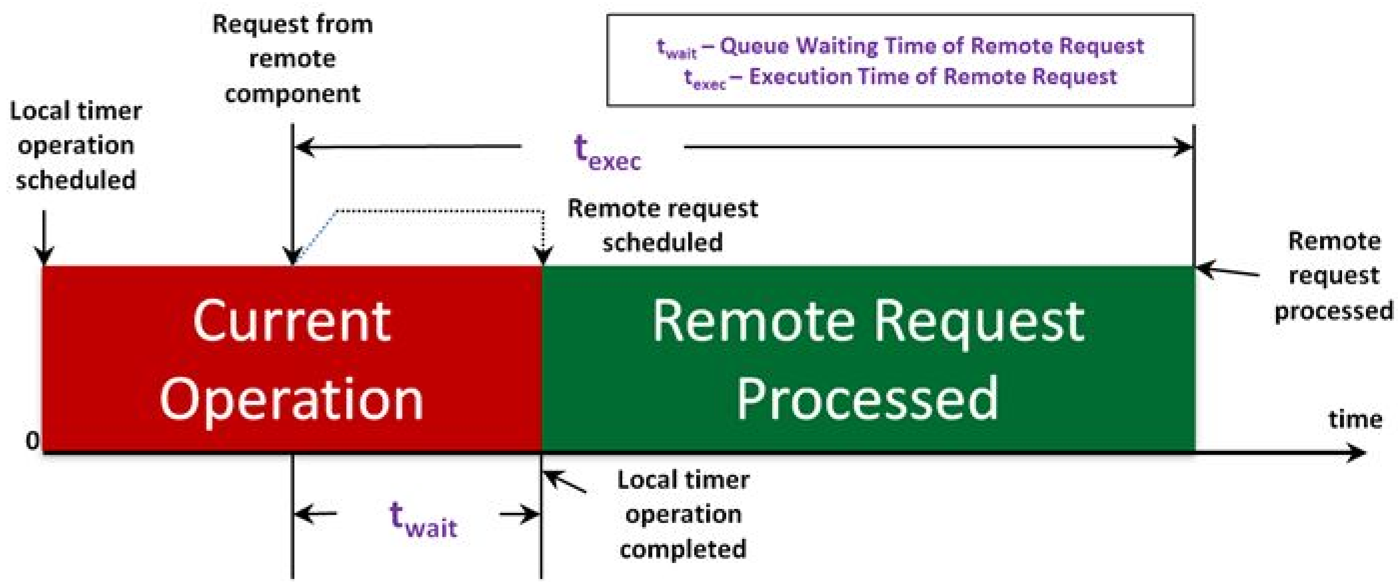
ROSMOD Deployment Infrastructure



Component Execution Semantics

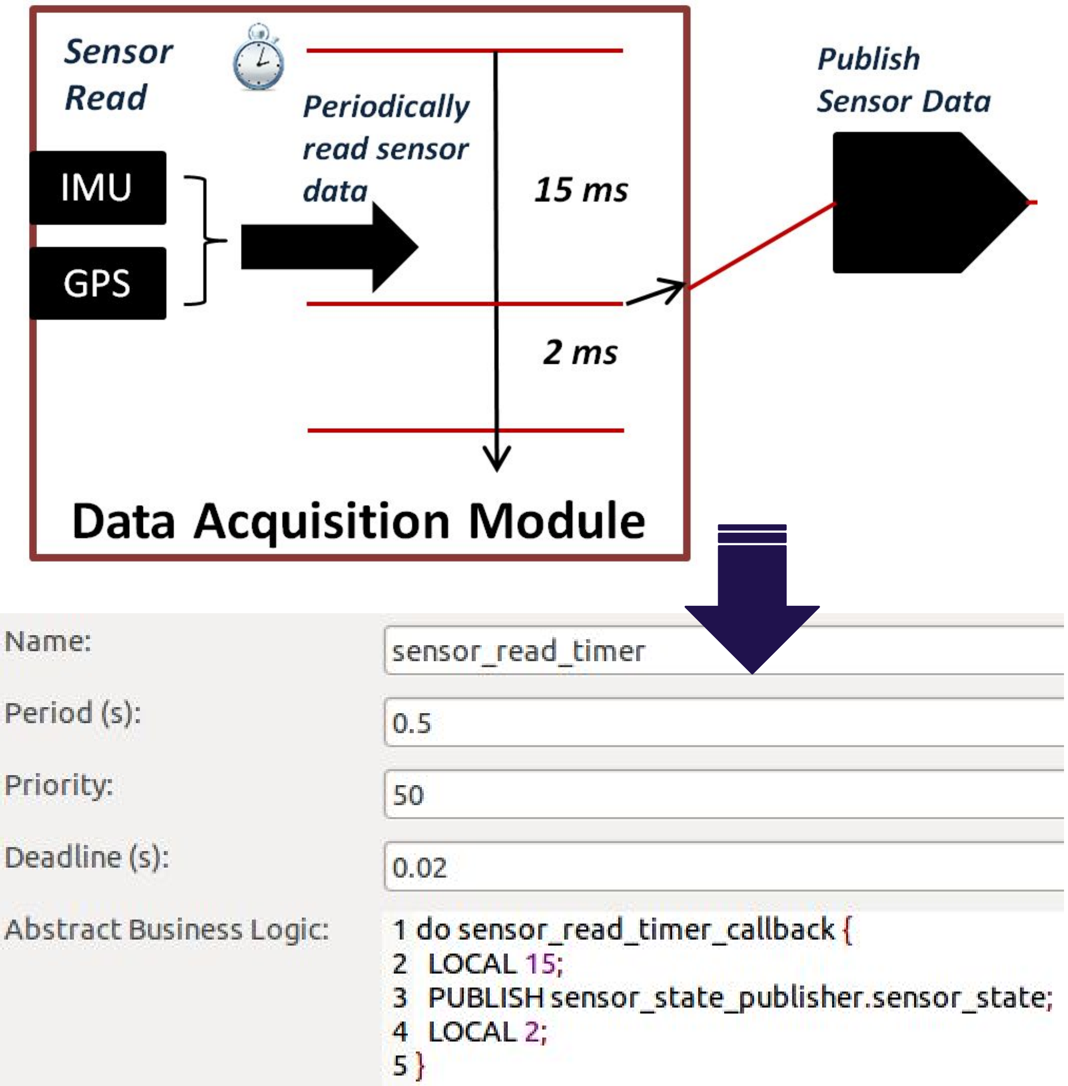
- A single component executor thread executes operation requests scheduled in the message queue
- Operation scheduling is non-preemptive i.e. the next request is processed only when the current operation is completed
- CHALLENGE:
    - Temporal Behavior of the composed system must meet end-to-end timing requirements
        - Deadlines, Trigger-to-Response times etc.





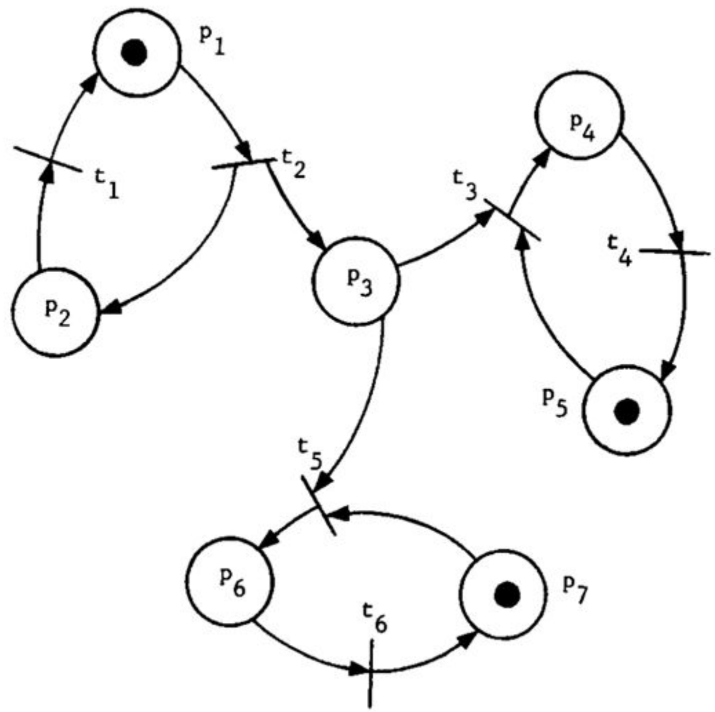
Component Business Logic - Analysis Integration

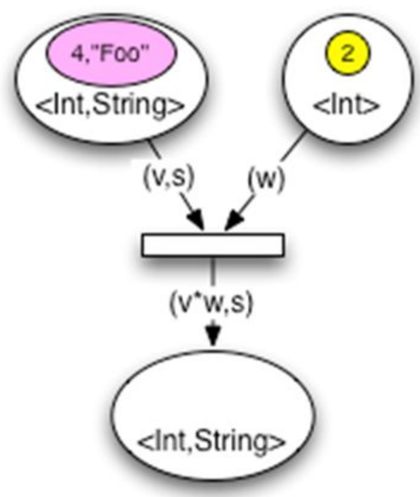
- <u>Challenge</u>
  - Operation Business logic: Piece of code executed when a component operation is scheduled
  - This code directly affects the behavior of components
  - It is not sufficient to annotate models of component operations with a single WCET
  - Need a finer grain model of temporal behavior
- <u>Approach</u>
  - Component Operations are modeled and represented as a sequence of timed steps - each step is annotated with a WCET



Petri Nets and CPN

- Formal model of information flow
- Places
    - Places contain tokens
    - Input Places and Output Places
- Transitions
    - Transitions represent events
    - Firing rules - input places with tokens
- Petri net Execution - Token Movement
- Modeling Dynamic System Behaviors
    - Concurrency, Synchronization and Resource Sharing
- Colored Petri Nets
    - Tokens can be complex typed data structures
    - Richer constraints on transitions
    - Compact hierarchical description
    - CPN Tools for simulation and state space analysis *[Ratzer et al., 2003]*





Colored Petri Net-based Timing Analysis

- Challenge
  - How to design and construct an extensible, scalable timing analysis model for component-based DRE systems?
- Approach
  - Design and implement a CPN-based timing analysis model
    - Capture structure and behavior of component assembly
    - Capture business logic of component operations (steps)
    - Capture timing properties of steps in each operations
- Preliminary Results
  - Developed an extensible CPN analysis model
  - Performed bounded *state space analysis* on a variety of DRE scenarios
    - Detect timing violations e.g. deadline miss, deadlocks etc.
    - Estimate worst-case trigger to response times
    - Estimate processor utilization

Analysis Results

- Worst-case Trigger to Response Time
  - Earliest Trigger
  - Latest Response
- CPU Utilization Estimation
  - CPU time used
  - CPU time available
- Deadline Violation Detection
  - Operation Execution Time
  - Operation Deadline
  - exec_time > deadline

```
(* Triggered Operation - Sensor_Read *)
val Trigger = (Search_cop_nodes "Sensor_Read" All_Completed);

(* Desired System Response - Completion of Servo_Control Operation *)
val Response = (Search_cop_nodes "Servo_Control" All_Completed);

(* Worst-case Trigger-to-Response Time Estimation *)
(calculate_response_time Trigger Response);
```

```
val Trigger =
  {comp_name="Data_Acquisition_Module",comp_node="UAV",op_dl=20000,
   op_et=17000,op_st=0,opname="Sensor_Read"}
  : {comp_name:string, comp_node:string, op_dl:int, op_et:int, op_st:int,
     opname:string}
val Response =
  {comp_name="Servo_Actuator",comp_node="UAV",op_dl=15000,op_et=77349,
   op_st=65000,opname="Servo_Control"}
  : {comp_name:string, comp_node:string, op_dl:int, op_et:int, op_st:int,
     opname:string}
val it = 77349 : int
```

```
val cpu_time_used_us = 1469280.0 : real
val cpu_time_available_us = 10000000.0 : real
val cpu_utilization = 14.6928 : real
```

```
(* Amout of CPU time utilized by component threads *)
val cpu_time_used_us = Real.fromInt (compute_requirement Completed_Operations);

(* Amount of CPU time available for the component thread execution *)
val cpu_time_available_us = Real.fromInt clock_limit;

(* CPU Utilization Estimation *)
val cpu_utilization = (cpu_time_used_us/cpu_time_available_us)*100.0;
```

```
val first_deadline_violation =
  ["Operation: Control_Loop; Execution Time: 52025 us; Deadline: 50000 us"]
  : string list
```

```
val first_deadline_violation = print_dl [hd (deadline_violation_check Completed_Operations)];
```

ASAP & Scalability Results

- Challenge
  - How to improve the efficiency and speed of state space generation and leverage advanced analysis and memory management techniques
- Approach
  - ASAP CPN Analysis Tool - Integrate our CPN analysis model with ASAP platform and apply advanced analysis methods
- Preliminary Results
  - Combat State Space Explosion - Sweep-line method
  - Safety Properties, Deadlock, Liveness, Boundedness, ...
  - On-the-fly efficient verification of system constraints
  - Time taken to generate state space is greatly reduced by applying ASAP memory optimization, hashing methods, and on-the-fly checking

Model	States	CPN Tools	ASAP	Speed-up
50 component DREMS sample	124K	846	211	4.01
100 component DREMS sample	485K	2,160	576	3.75