Towards Introducing Asynchronous Tasks to an FRP Language for Small-Scale Embedded Systems

Akihiko Yokoyama, Sosuke Moriguchi and Takuo Watanabe
Department of Computer Science
Tokyo Institute of Technology

Overview

Objective

- Support for the **heavy task** execution in an FRP language for small scale embedded systems
  - **Heavy tasks**: tasks that may degrade the responsiveness of the systems

Proposed Method

- A simple asynchronous task execution mechanism that supports mutual exclusions of computational resources
  - Future types
  - Tasknodes: special time-varying values for heavy tasks
  - Task resources: abstractions of computational resources for heavy tasks

Contribution

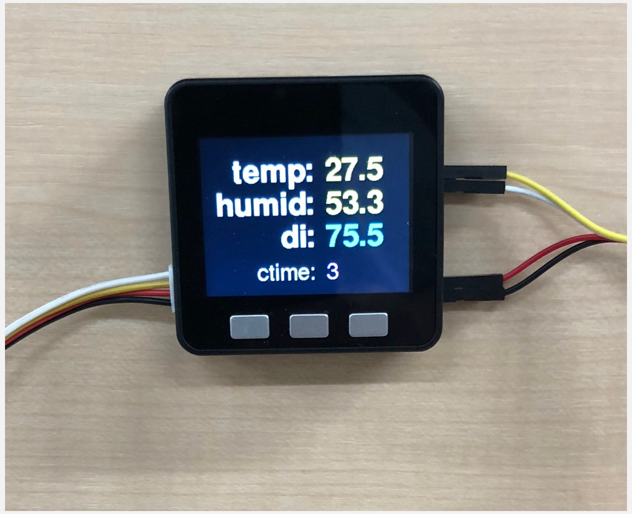
- Embedding heavy task executions to reactive programs naturally.

Emfrp [Sawada et al. 2016]

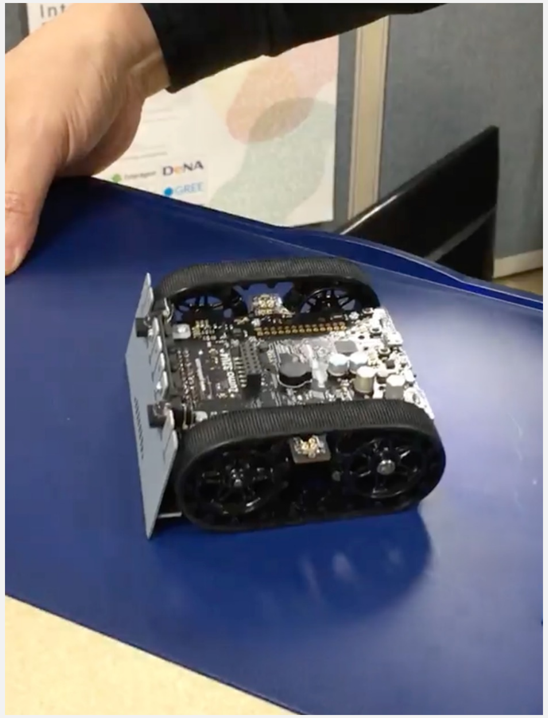
- Statically-Typed Pure FRP Language
  - Designed for small-scale (resource-constrained) embedded systems
    - Ex) AVR, STM32, ESP32 etc.
  - Statically-determined runtime memory size
  - Nodes represent time-varying values.
  - The values in the previous moment can be referred by **@last** operator

```
module FunController
in  tmp: Float, hmd: Float
out fan: Bool
use Std

node di = 0.81 * tmp +
          0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
node init[False] fan = di >= th
node th = 75.0 + if fan@last then -0.5 else 0.5
```

Emfrp module to control a fan by discomfort index



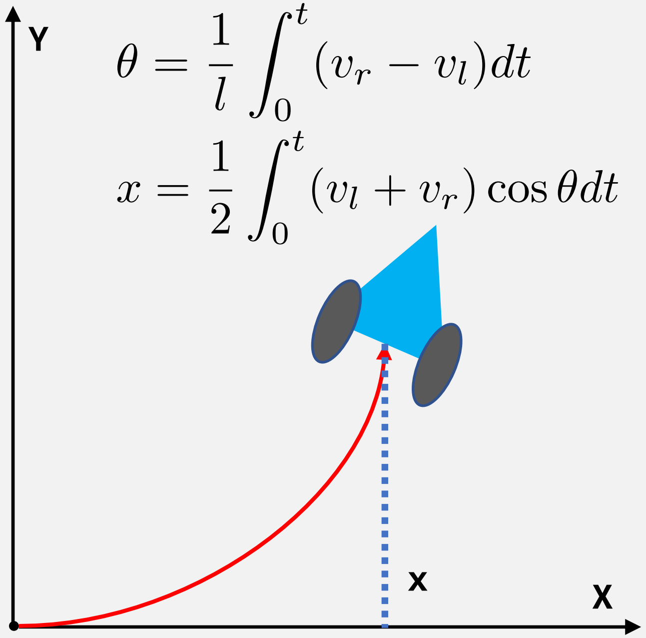
M5 Stack (CPU: ESP32 240MHz, Flash: 16MB, RAM: 520KB)


Zumo (CPU: ATmega328p 16MHz, Flash: 32KB, RAM: 2KB)

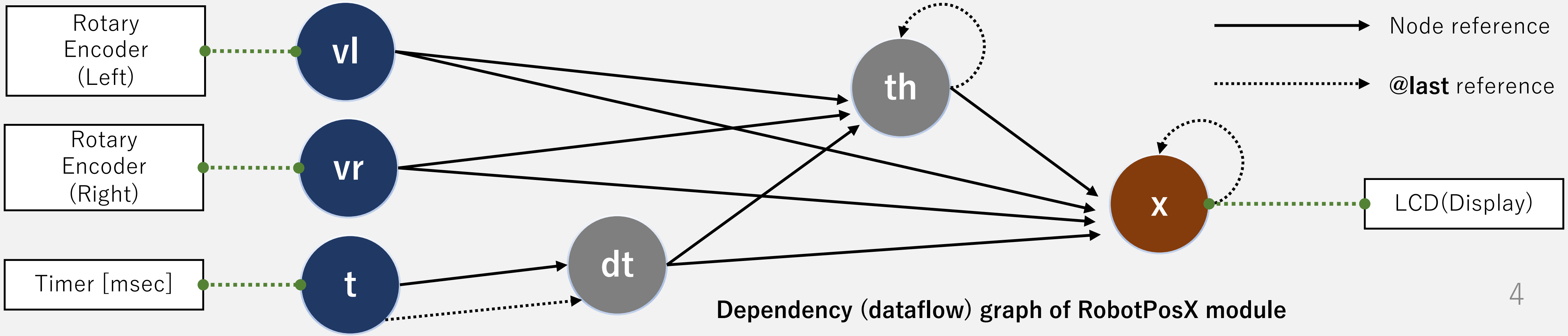
Example: Position of a Robot in Emfrp

$$\theta = \frac{1}{l}\int_0^t (v_r - v_l)dt$$

$$x = \frac{1}{2}\int_0^t (v_l + v_r)\cos\theta dt$$

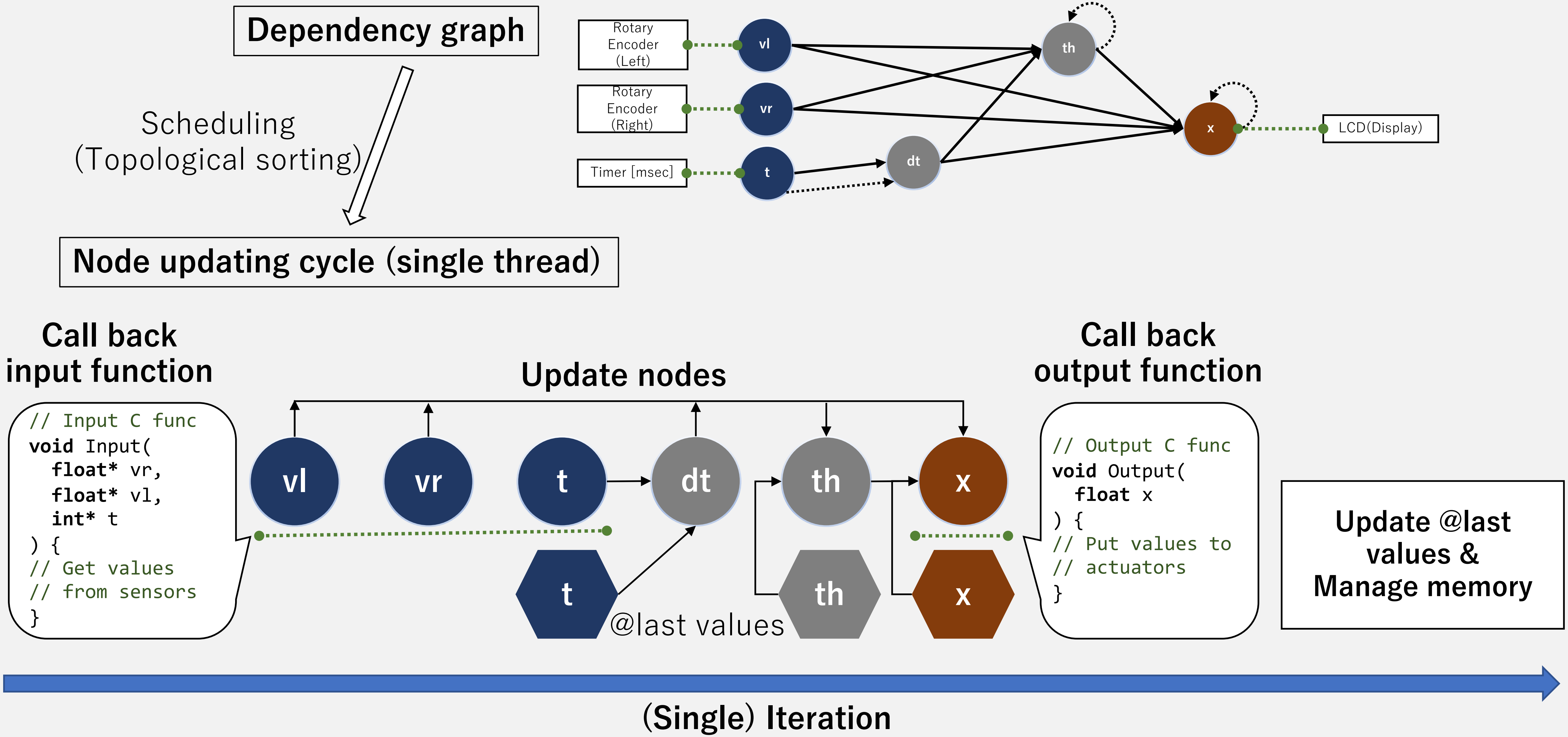



```
# RobotPosX.mfrp
module RobotPosX   # Module Name
in   vl : Float,   # Left velocity [m/sec]
     vr : Float,   # Right velocity [m/sec]
     t(0) : Int    # Elapsed time [msec]
out x : Float,     # X-coordinates [m]
use Std, Params    # Import library

# Intermediate nodes
node dt = (t - t@last) / 1000.0
node init[0.0] theta = theta@last + (vr - vl) * dt / l

# Output node (X-coordinates)
node init[0.0] x = x@last + (vr + vl) * cos(theta) * dt / 2.0
```



Dependency (dataflow) graph of RobotPosX module

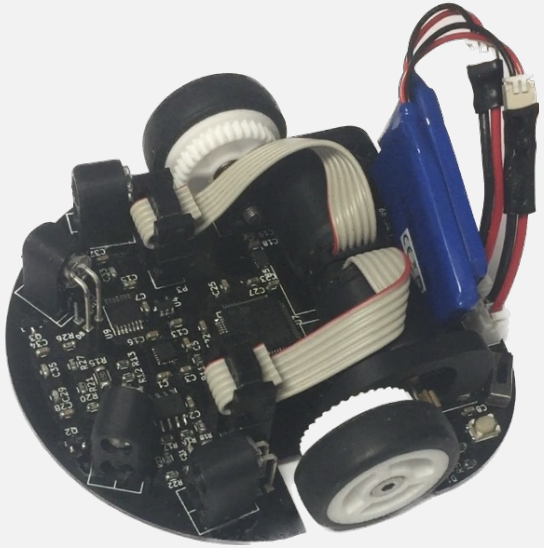
Execution Model of Emfrp

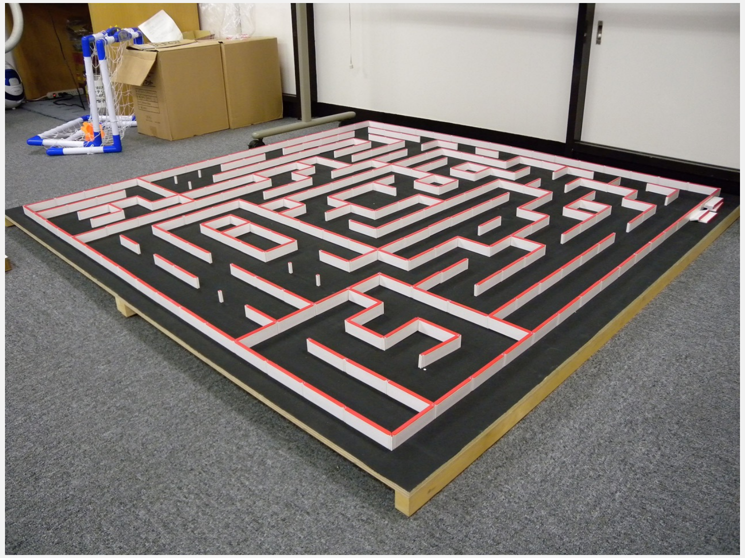


Motivating Example: Exploration Robot

Micromouse

- A robot competition
- Small-scale embedded systems
  - Microcontroller (STM32 etc.)
  - Sensors
    - Infrared sensors
    - Rotary encoders
    - IMU (accelerometer, gyro sensor)
  - Actuators
    - Motors
    - LEDs
- Computation on **graph structures**



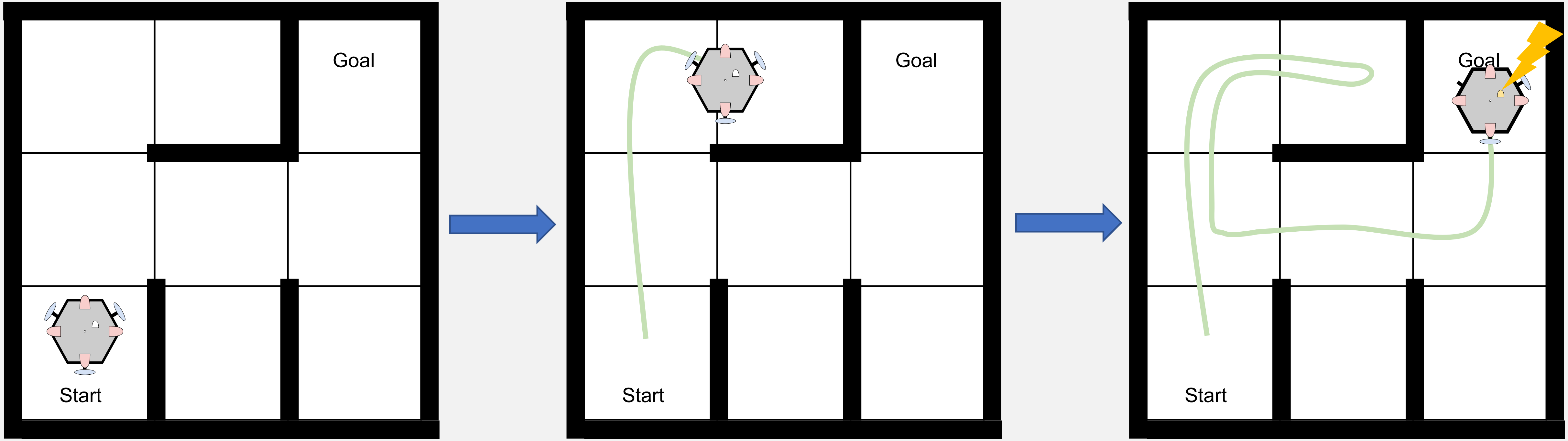


https://en.wikipedia.org/wiki/Micromouse

Motivating Example: Exploration Robot

Problem Setting

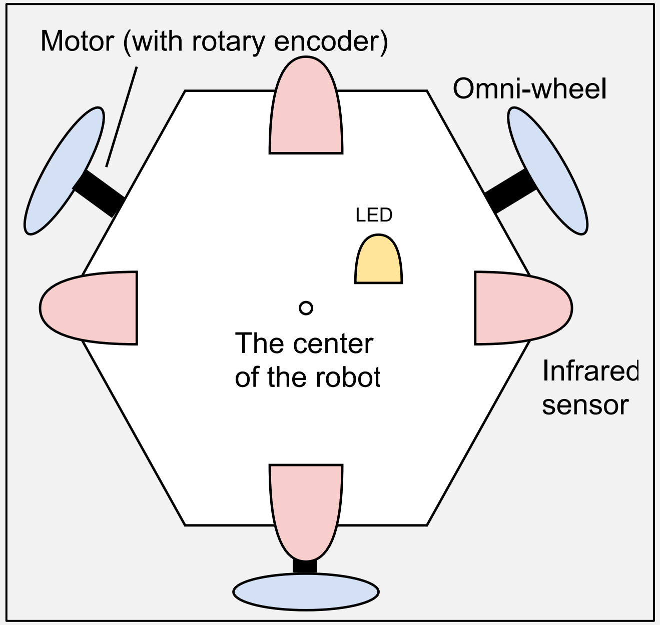
- Simplified version of micromouse.
- Starting from the start, the robot explores the maze autonomously and turns on the LED when it reaches the goal.



Motivating Example: Exploration Robot

Hardware Specifications of the Exploration Robot

- Omni-directional mobile vehicle
- Sensors
  - Wall sensor: Infrared sensor x4
  - Velocity sensor: rotary encoder x3
- Actuators
  - Motor with omni-wheel x3
  - LED x1



https://en.wikipedia.org/wiki/Omni_wheel

Motivating Example: Exploration Robot

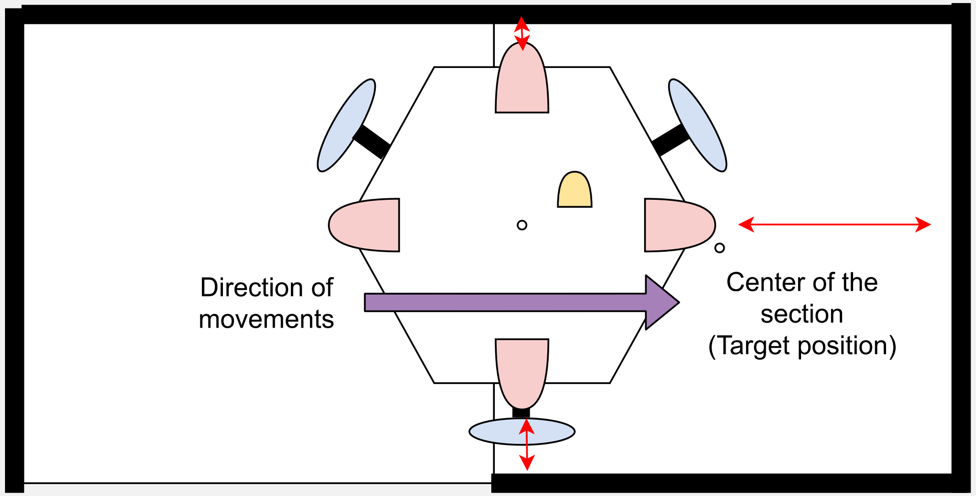
Naive (Standard) Exploration

- Repeat the following actions until reaching the goal.
  - A: Moving the robot to the target section and Stop.
  - B: Recording the wall information.
  - C: Calculating the next destination by a graph algorithm (e.g., A*).

Improved Exploration

- Wall information can be obtained before reaching the target position.

→ If A and (B, C) can be executed concurrently, the latency for exploring is reduced.



Heavy Tasks

Heavy tasks

- Computations not required to be executed every iteration, but are relatively **time-consuming**
  - Heavy task execution during iterations degrades responsiveness of the whole system.
- Heavy tasks are assumed to be operations on somewhat complex data structures.
  - E.g., graph algorithms, parsing, etc.
  - Updating or searching “maze” in the exploration robot example.

Task resources

- The objects handled by the heavy task
  - E.g., the data structures representing “maze”

Restrictions in Emfrp

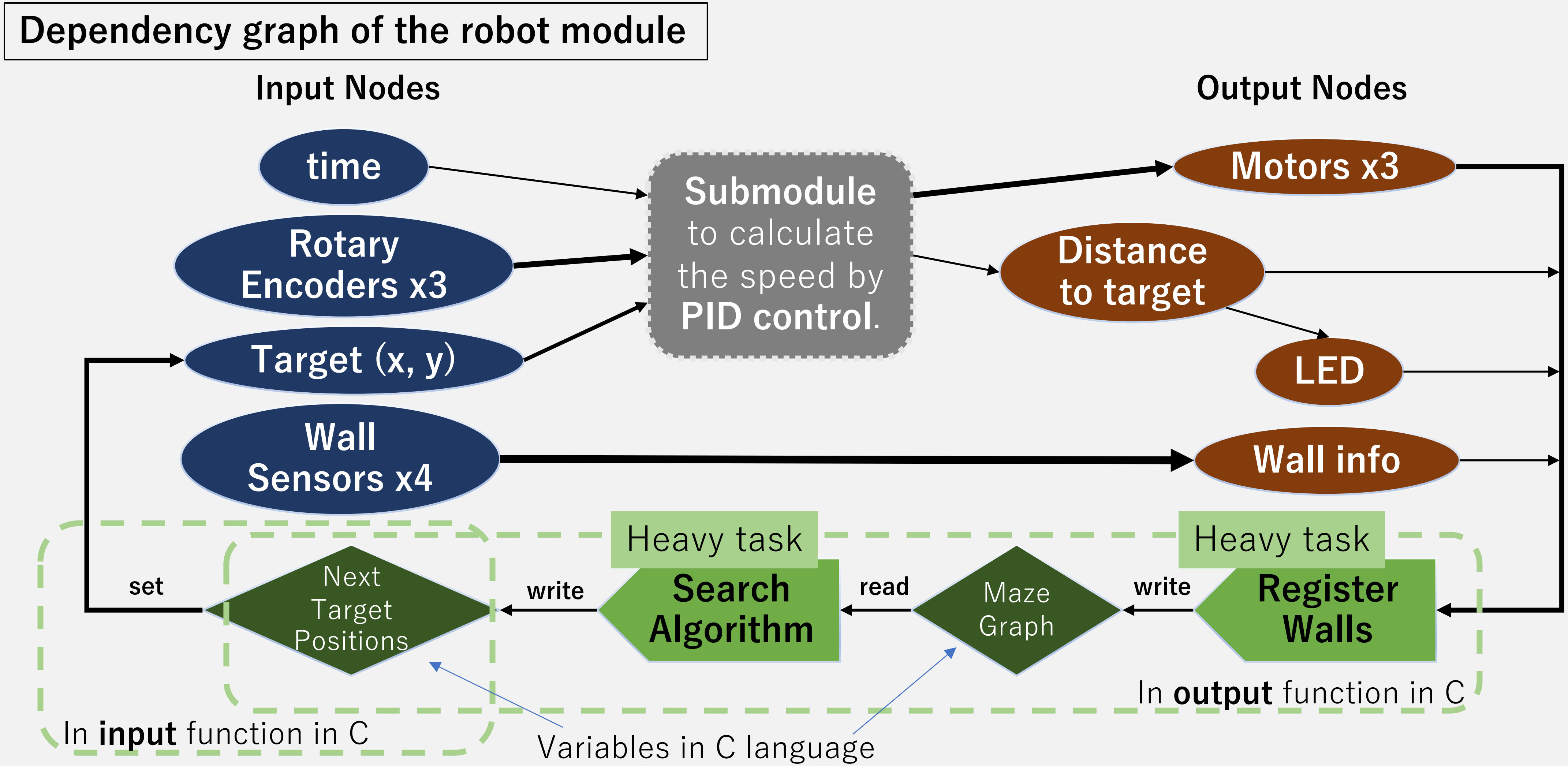
- Time-varying values are not first-class
  → Preventing space / time leak

- Recursive data types and recursive functions are prohibited.
  → Determination of memory used at runtime

- The data structures representing “maze” and related operations cannot be written in Emfrp.
  → Requires foreign (C language) function calls.

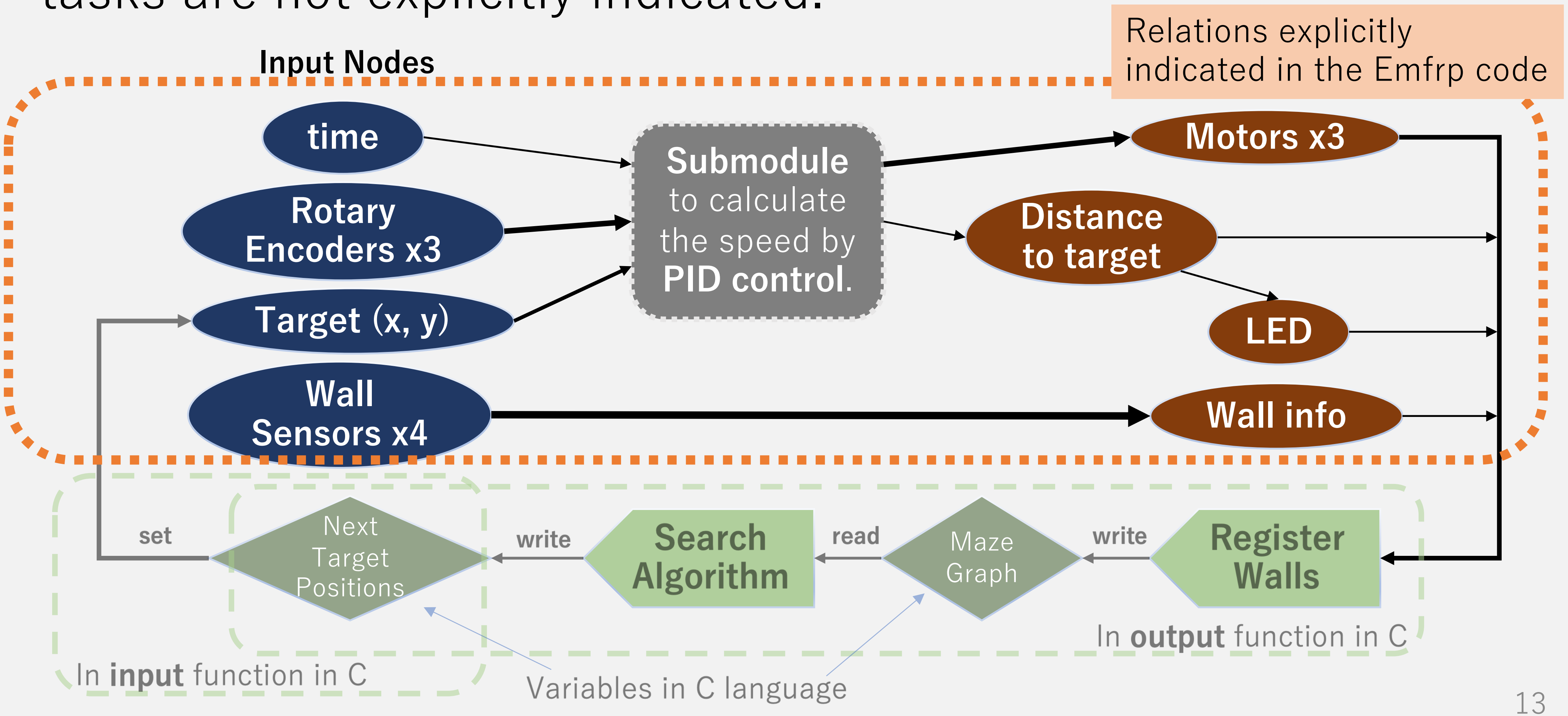
Design of the robot in original Emfrp

Heavy-task execution during feedback from output to input.



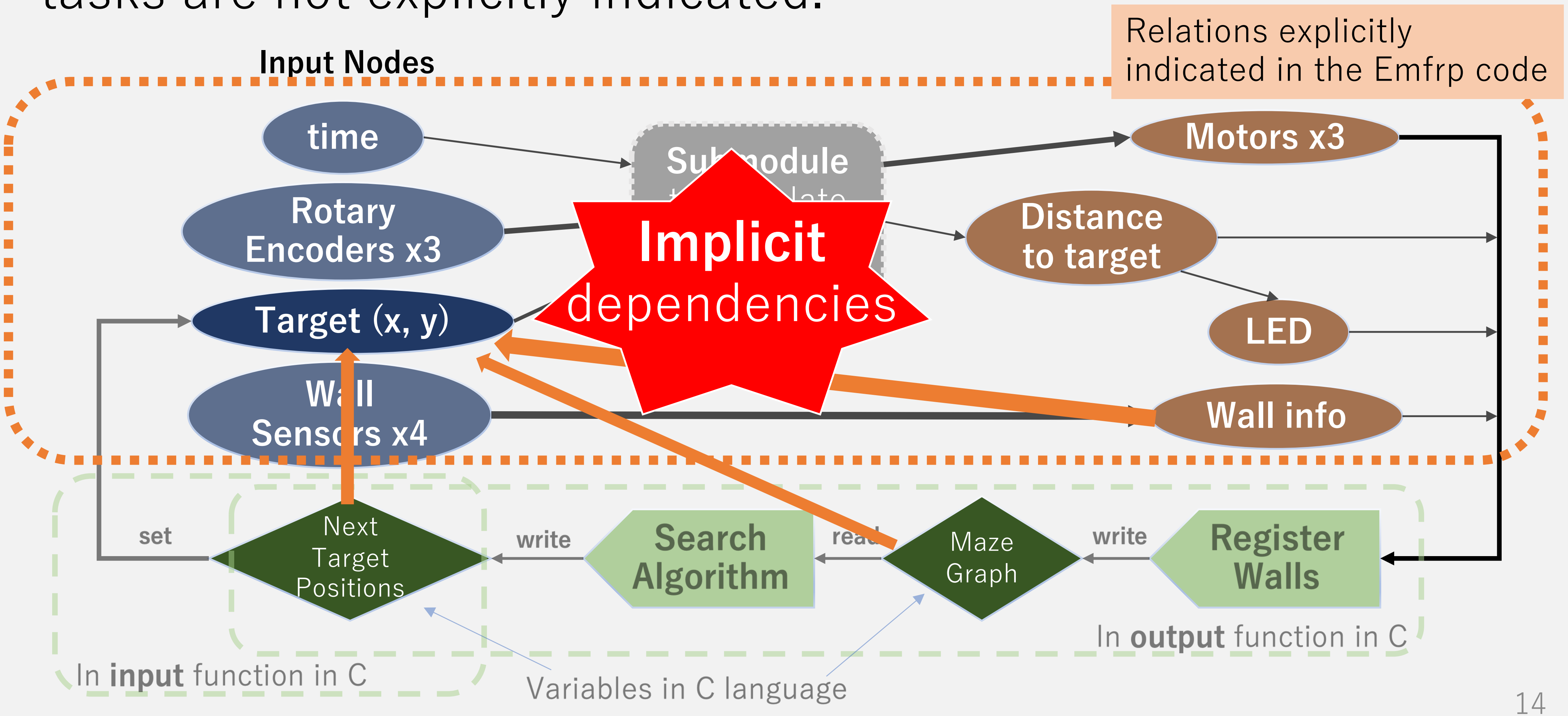
Problem (P2 in the paper) of the robot module

The dependencies and/or the data structures subject to heavy tasks are not explicitly indicated.



Problem (P2 in the paper) of the robot module

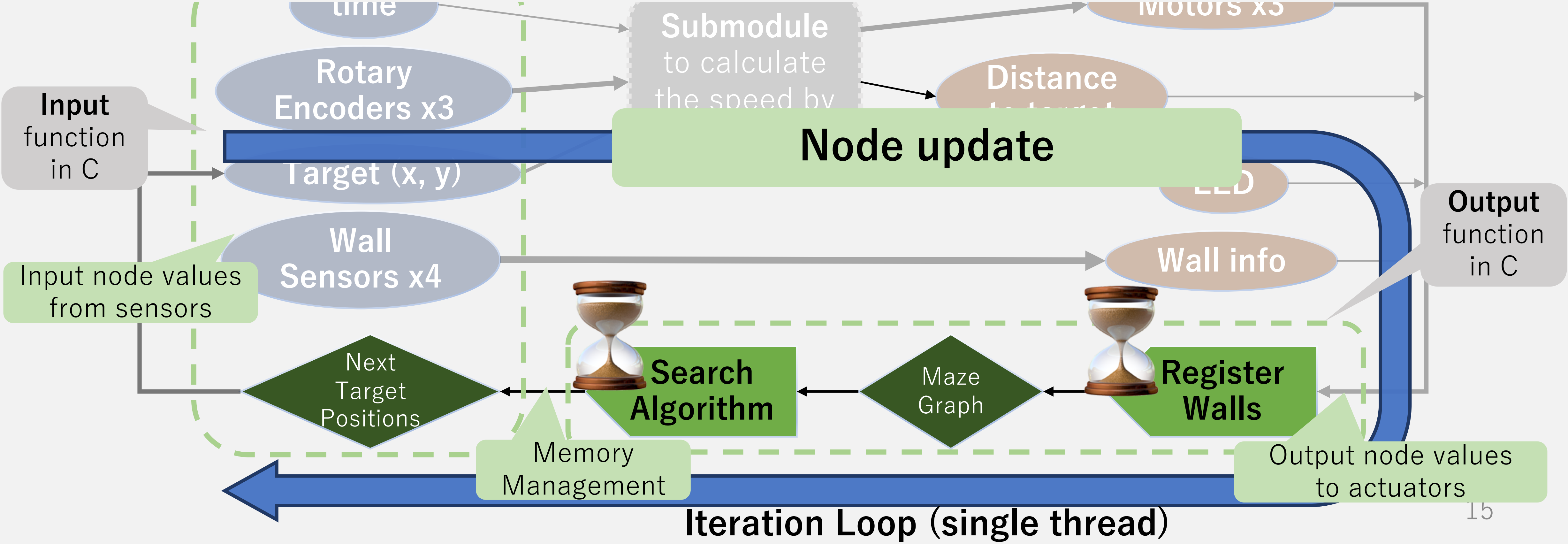
The dependencies and/or the data structures subject to heavy tasks are not explicitly indicated.



Problem (P1 in the paper) of the robot module

The sequential execution of heavy tasks in the output function makes other reactive behaviors less responsive.

- Due to execution model of original Emfrp.
- **Reactive Thread Hijacking Problem (RTHP)** [Van den Vonder et al. 2020]

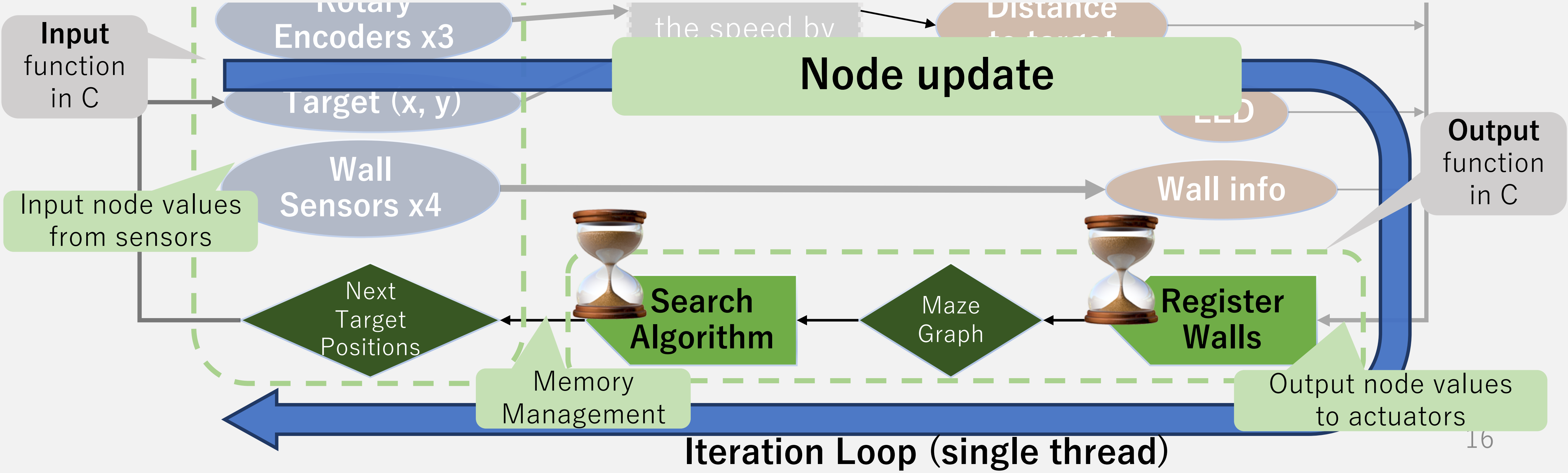


Problem (P1 in the paper) of the robot module

The sequential execution of heavy tasks in the output function makes other reactive behaviors less responsive.

- Due to execution model of original Emfrp.
- **Reactive Thread Hijacking Problem (RTHP)** [Van den Vonder et al. 2020]

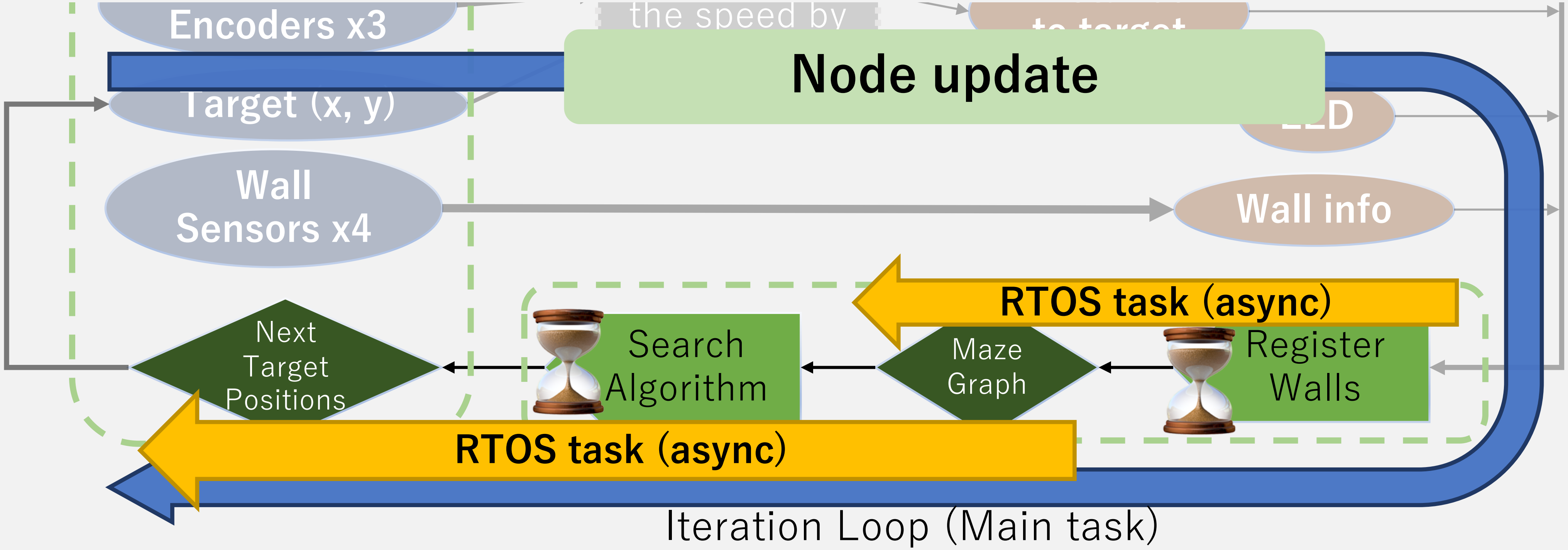
→ **Cannot implement the improved search.**



Problem (P3 in the paper) of the robot module

Solution? of P1: Cooperation with concurrent execution libraries.

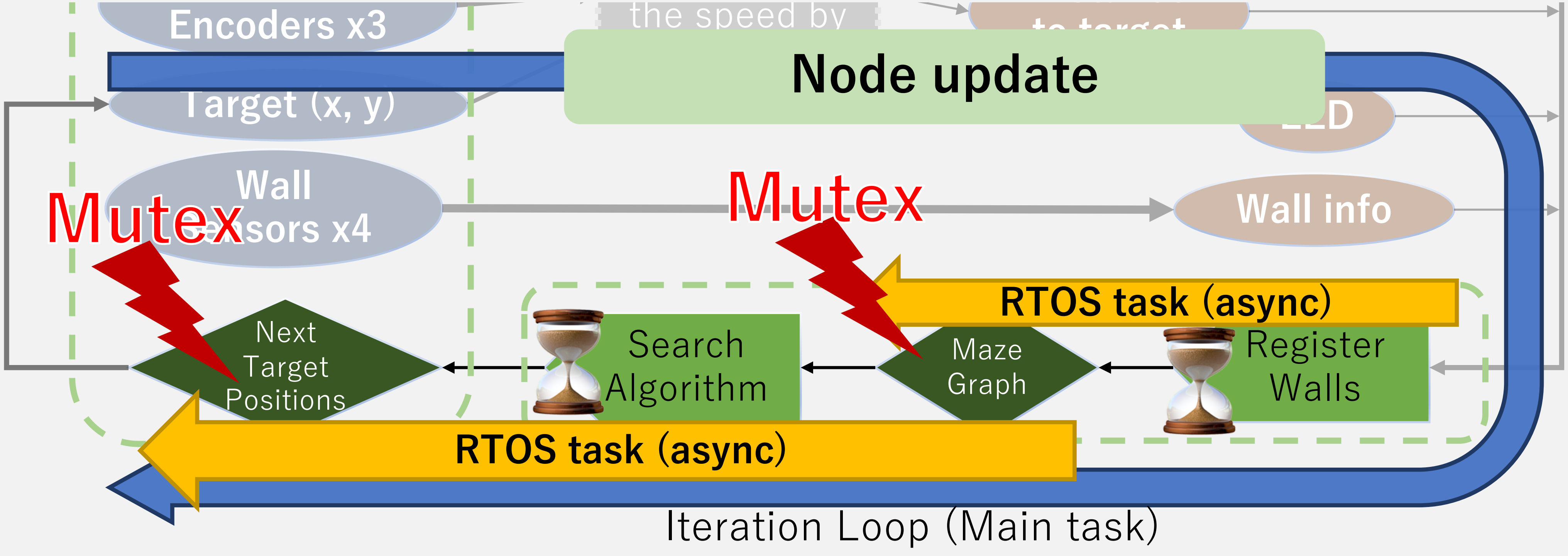
- E.g., FreeRTOS's tasks
- Reactive computations (iteration loop) and heavy tasks can be executed concurrently.



Problem (P3 in the paper) of the robot module

Solution? of P1: Cooperation with concurrent execution libraries.

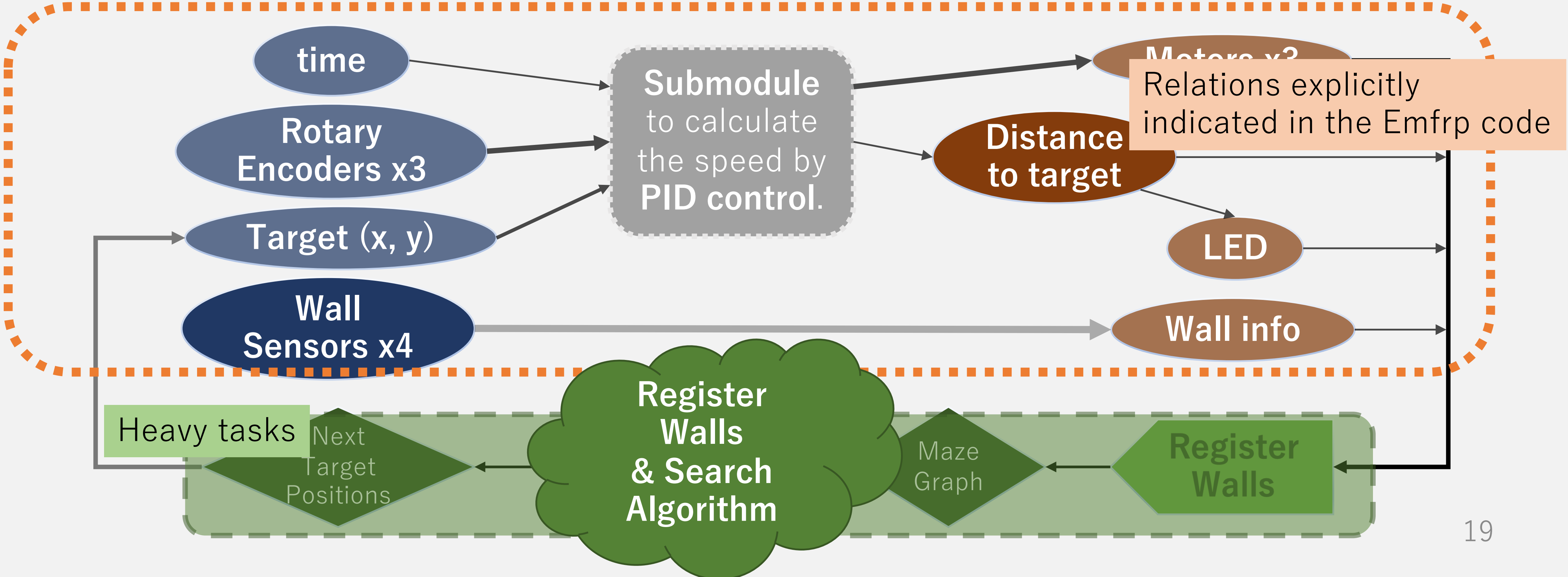
- E.g., FreeRTOS's tasks
- Reactive computations (iteration loop) and heavy tasks can be executed concurrently.
- **(P3) Mutual exclusions** for task resources (shared variables) is required.
  → The advantage of Emfrp (or FRP) is lost.



Our Approach

We want to make heavy tasks and task resources **explicit** in the scope of Emfrp.

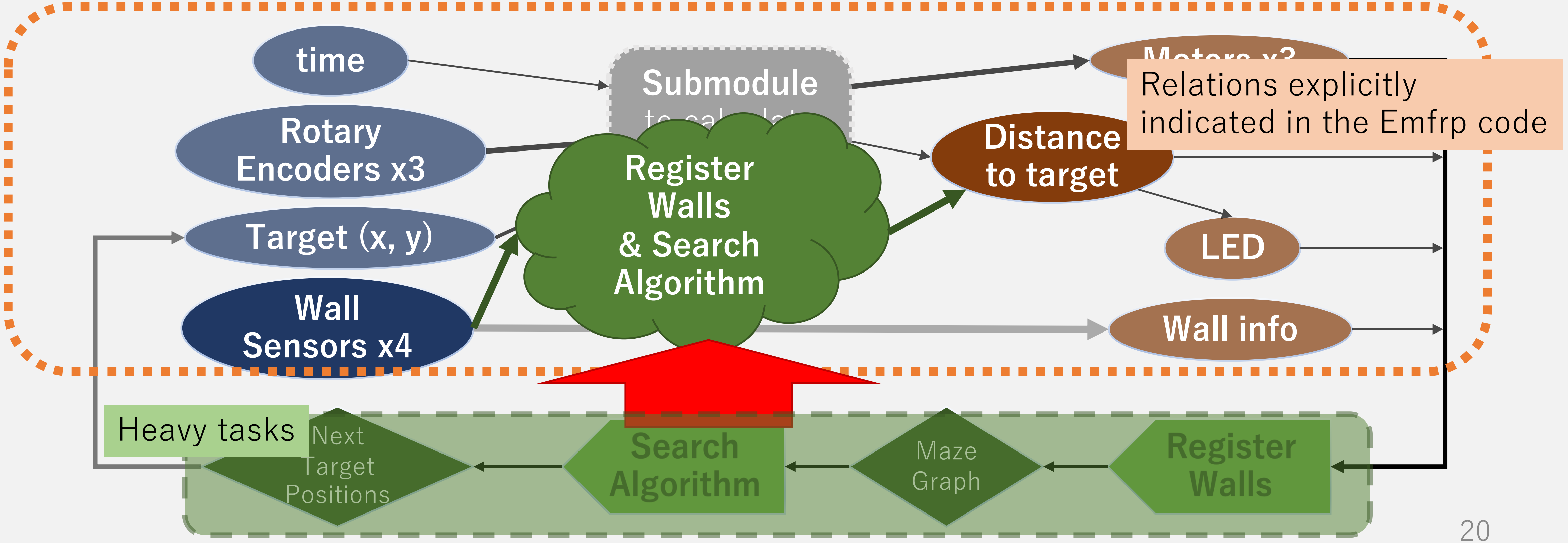
- Enhance **readability** and **maintainability**.



Our Approach

We want to make heavy tasks and task resources **explicit** in the scope of Emfrp.

- Enhance **readability** and **maintainability**.

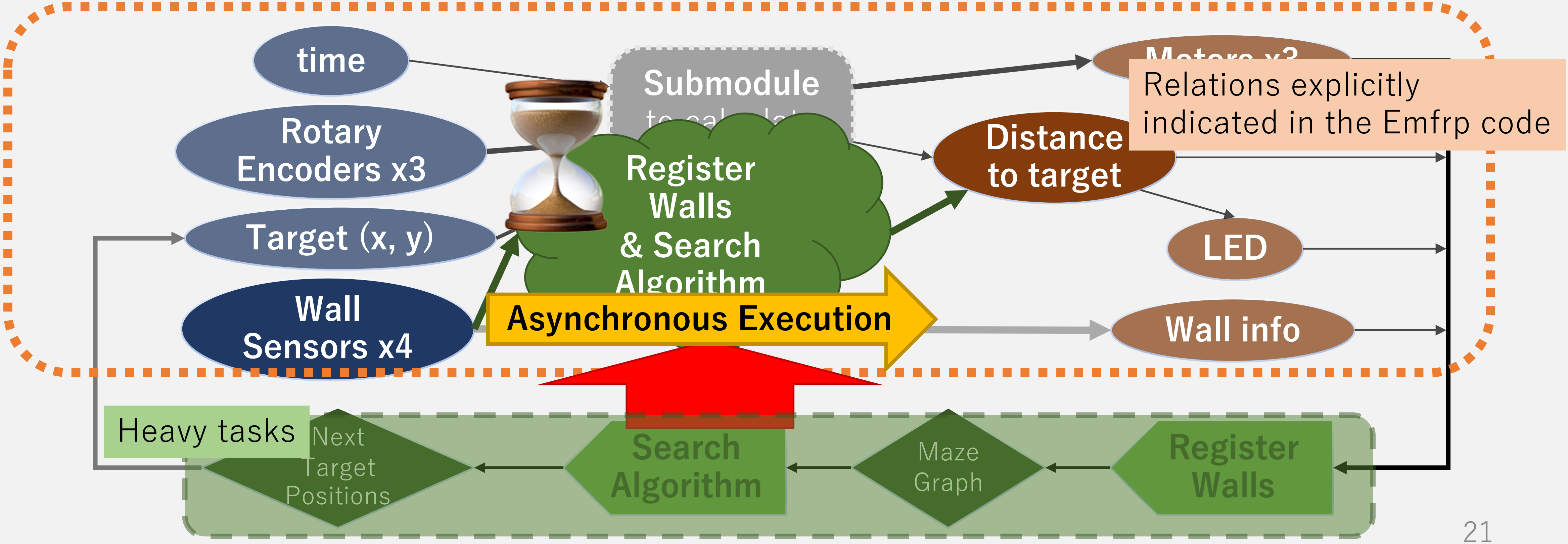


Our Approach

We want to execute heavy tasks **asynchronously** to prevent RTHP.

- **Responsiveness** of the whole system is not degraded.

→ **Asynchronous executions** and **Future types**



Proposed method

- Language Extensions
  - Future types
  - Definitions of tasks and task resources
  - Tasknodes: special nodes invoking heavy tasks

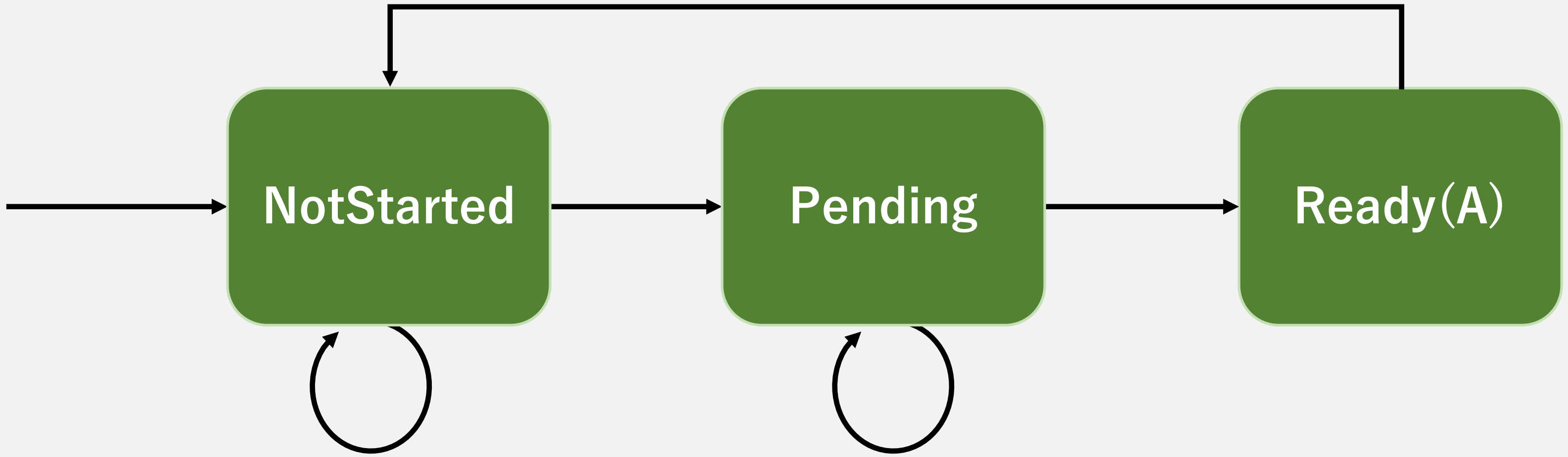
- Runtime Extensions
  - Inserting an asynchronous execution phase of a task into each iteration
  - Simple task scheduler for resource-constrained environments with mutual exclusion of task resources

Language Extensions: Future types

- Future types represent the state of the task computation.

```
type Future[A] =
  | Ready(A)     # Just after the task is completed.
  | Pending      # Waiting for the task completion
  | NotStarted   # The task is not issued.
```

State transitions of the value of future types



Language Extensions: Tasks and Task resources

- Definitions of task resources and tasks
  - Task resources are compiled to skeleton codes of C structure
  - Tasks are compiled to skeleton codes of C function

Name of task resources

Result of the task

```
resource MazeGraph {
  # Task to record wall information of section (u, v) in a MazeGraph instance
  task RegisterSection :
    (u: Int, v: Int,
     n: Bool, e: Bool, s: Bool, w: Bool) -> (h: Unit) / write

  # Task to compute the next section (next_u,next_v) to go from (u,v) to the goal
  task CalcNextSection :
    (u: Int, v: Int, goal_u: Int, goal_v: Int)
        -> (next_u: Int, next_v: Int) / read
}
```

Hint for mutual exclusion

Input parameters of the task

Language Extensions: Tasks and Task resources

- Definitions of task resources and tasks
  - Task resource instance

```
resource MazeGraph {
  # Task to record wall information of section (u, v) in a MazeGraph instance
  task RegisterSection :
    (u: Int, v: Int,
     n: Bool, e: Bool, s:

  # Task to compute the n                                                          oal
  task CalcNextSection :
    (u: Int, v: Int, goal
        -> (next_u: Int,
}
```

```
# Toplevel module
module MazeRunner
in mg : resource MazeGraph
   sn: Int, se: Int, … time(0): Int
out duty1:Int, …
use Std, Resources

node move_dir = …
…
```

Passing the task resource instance from C code (activation function)

Language Extensions: Tasknodes

Tasknode

- Special node that ties the heavy task to the future type node.
  - Issue the task when the condition is satisfied.
    - Required that state of the task is NotStarted.
  - The parameters of the issued task are snapshots of node values.

Node name

Future type

Task name

```
tasknode next : Future[(Int, Int)] =
  CalcNextSection(to_u, to_v, G_U, G_V)
    with mg at wait(finish_register)
```

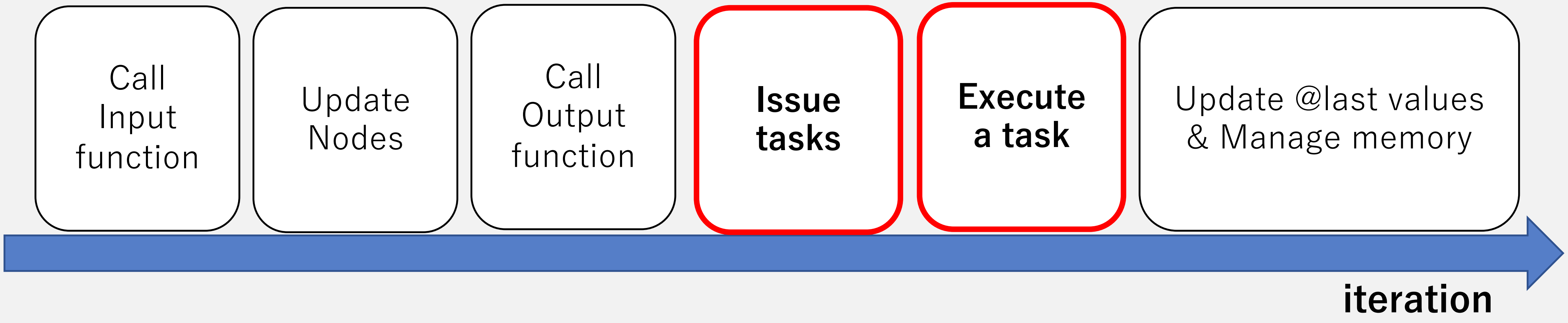
Instance of the task resource

The condition (timing) for issuing tasks

Parameter
(snapshots of node values)

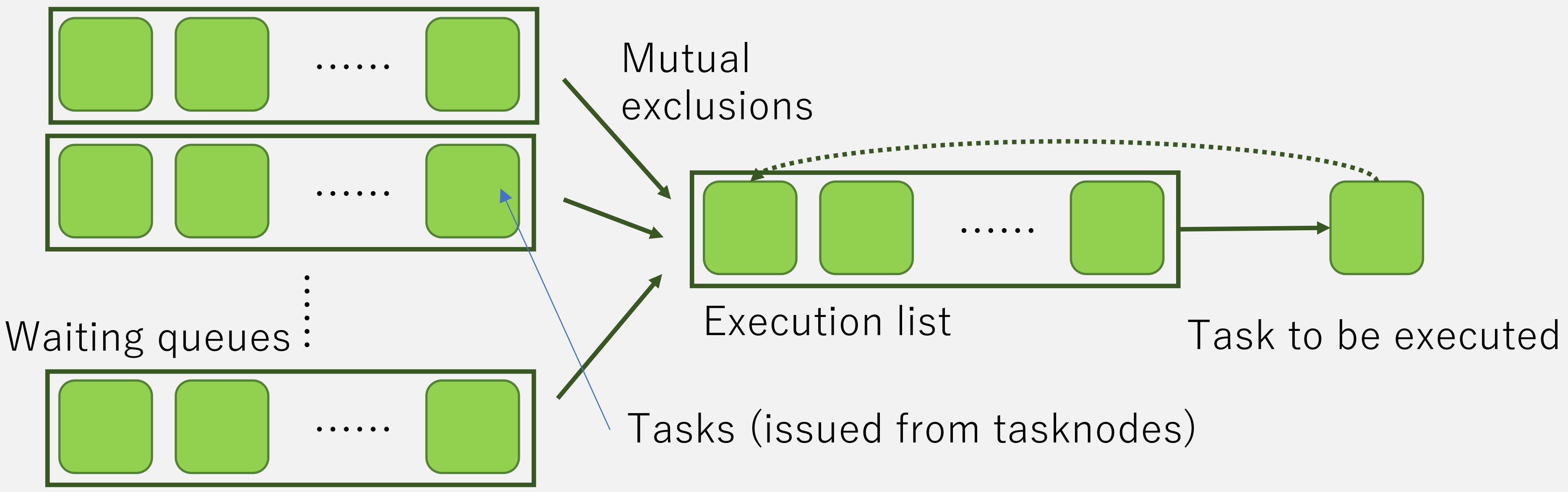
Runtime Extension: Iteration Loop

- Add phases related heavy tasks to iteration loop
  - Issuing tasks
  - Executing a task for N (milli)seconds
    - N is specified at compile time
    - One task execution per iteration (Round-robin scheduling)
    - Preemptive task execute using timer interrupts or RTOS tasks
      - Traditinal context switch techniques



Runtime Extension: Task Scheduling

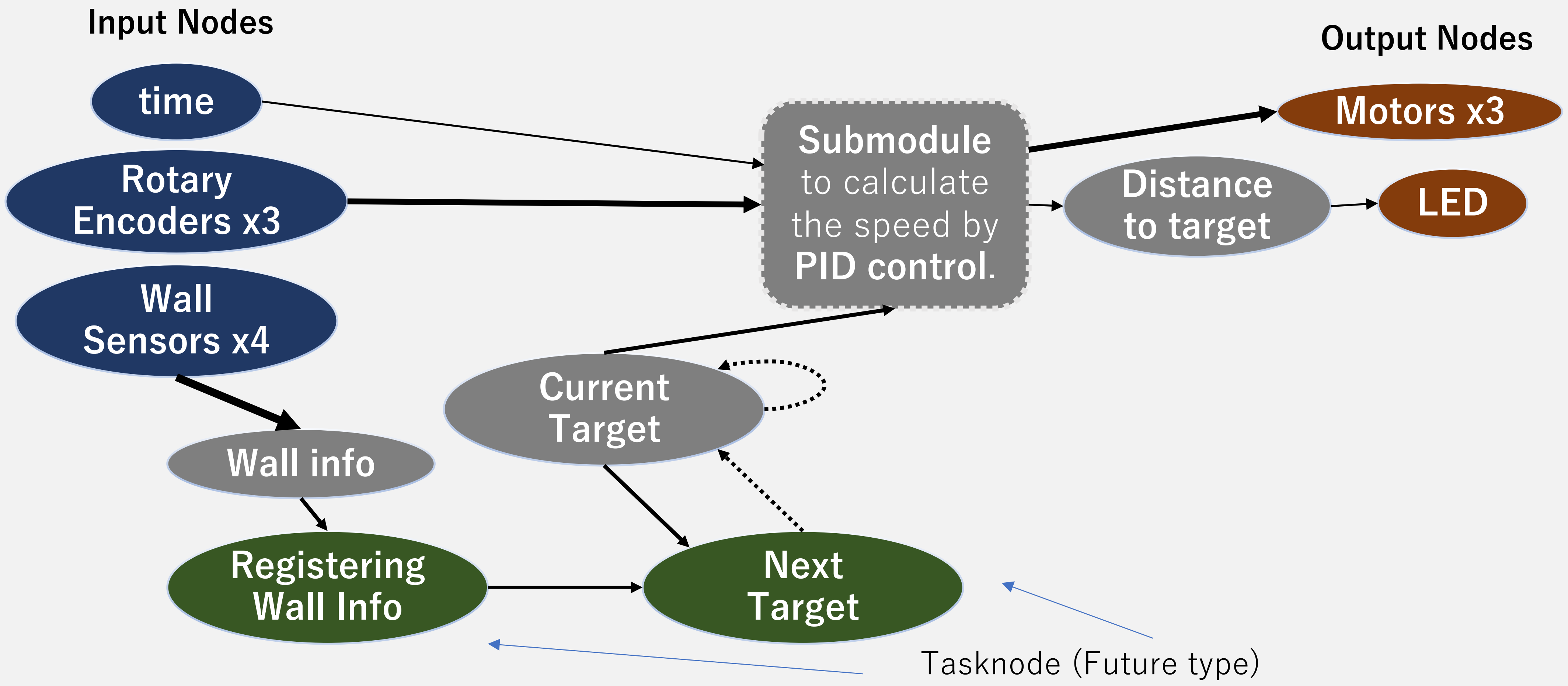
- Round-robin scheduling
- Simple task execution management algorithm
  - Designed for single-core microcontrollers
  - Mutual exclusion of task resources using **`read`**/**`write`** on tasks
  - Execution list, Waiting queues (per task resource instance)
    - Maximum queue length can be conservatively estimated.



Design of the robot in Extended Emfrp

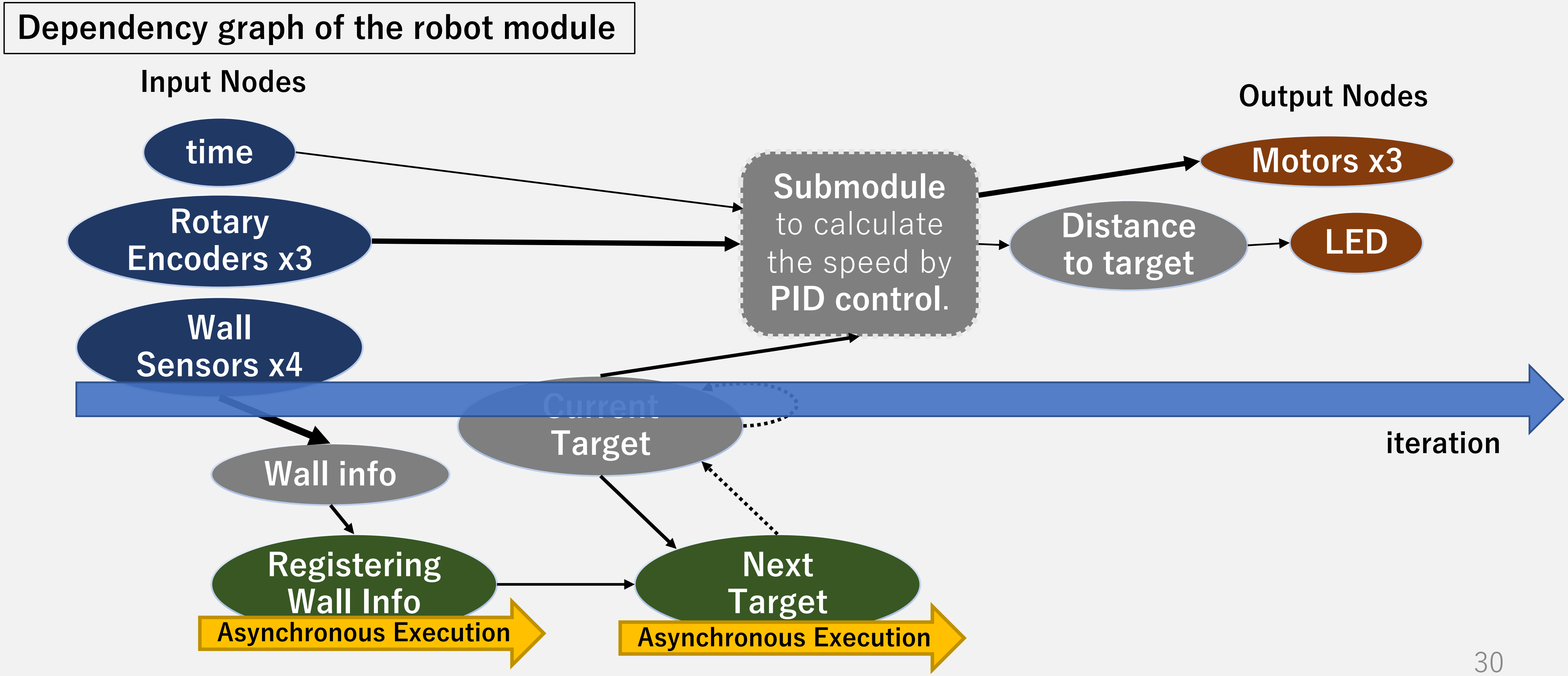
Heavy-task execution with future nodes.

Dependency graph of the robot module



Design of the robot in Extended Emfrp

Heavy-task execution with future nodes.



Related works

- Actor-Reactor model [Van den Vonder et al. 2020]
  - A method to separate and modularize reactive (reactor) and procedural (actor) behavior descriptions.
  - Actors (heavy tasks) and reactors are in single language (Stella).
    - Dependencies are explicitly defined in the single language.
  - They also advocated Reactive Thread Hijacking Problem.

- Lustre with Futures [Cohen et al. 2012]
  - Lustre: a synchronous dataflow language
  - Their future types enables concurrency, pipelining and jitter control.
    - Compiled into Java's threads.
  - Not for heavy tasks defined as foreign (C language) functions.

Future Tasks

- Implementation and evaluation of Emfrp compiler with proposed methods.

- Design of abstractions for task resources
  - Coarse-grained task resources are sufficient for current examples.
    - Heavy tasks are **computationally bound heavy tasks** in this presentation.

  - We need more fine-grained resource abstractions for I/O devices.
    - **I/O bound heavy tasks** will handle network or serial ports.

Conclusion

- We proposed an asynchronous task execution mechanism for Emfrp, an FRP language for small embedded systems.
- The mechanism enables the heavy task executions while keeping sufficient responsiveness.
- We showed the usefulness of the mechanism through the explanation of an non-trivial example.