小規模組込みシステム向けFRP言語における非同期タスク処理機構

東京工業大学 情報理工学院

横山陽彦　森口草介　渡部卓雄

PPL 2022 / 2022年3月7日

研究概要

目的

- 小規模組込みシステム向けFRP言語Emfrpにおける
  応答性を低下させない外部関数呼び出し機構

提案手法

- 非同期タスク処理機構の導入
  - tasknode定義，Future型，タスクが扱うリソースを明示する機構の導入による言語拡張
  - 小規模環境向け非同期タスク実行基盤の導入によるランタイム拡張

貢献 (提案)

- 時変値間の暗黙的な依存関係を導入しない外部関数呼び出し機構
- シングルコアマイコンでも動作可能な非同期タスク実行基盤

背景：関数リアクティブプログラミング

Functional Reactive Programming (FRP)

- リアクティブシステムの簡潔な記述を目指すプログラミングパラダイム
- 時変値を宣言的に組み合わせたデータフローの明示

リアクティブシステム

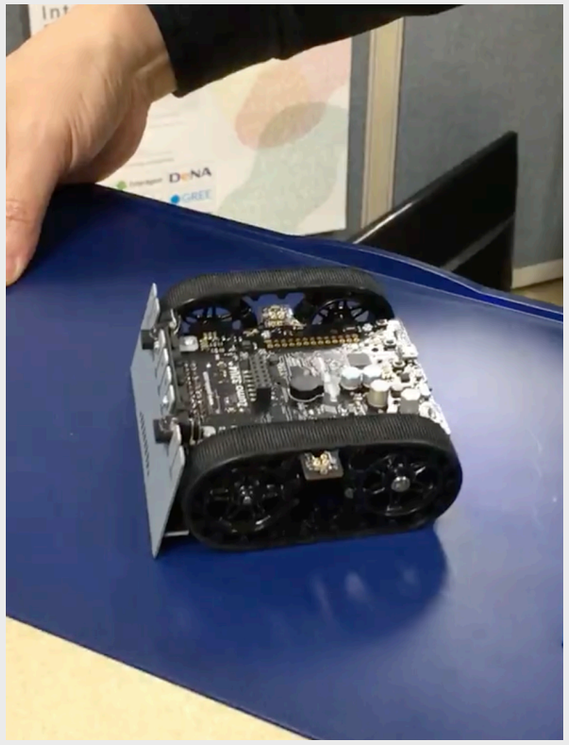
- 入力に対して内部状態を変化させながら応答するシステム
  - 例：GUI，組込みシステム，アニメーション等
- 手続き型言語による実装ではポーリングや割り込みを多用[Bainomugisha et al. 2013]
  - 変数間の値の伝搬が把握しづらい

時変値(time-varying value)

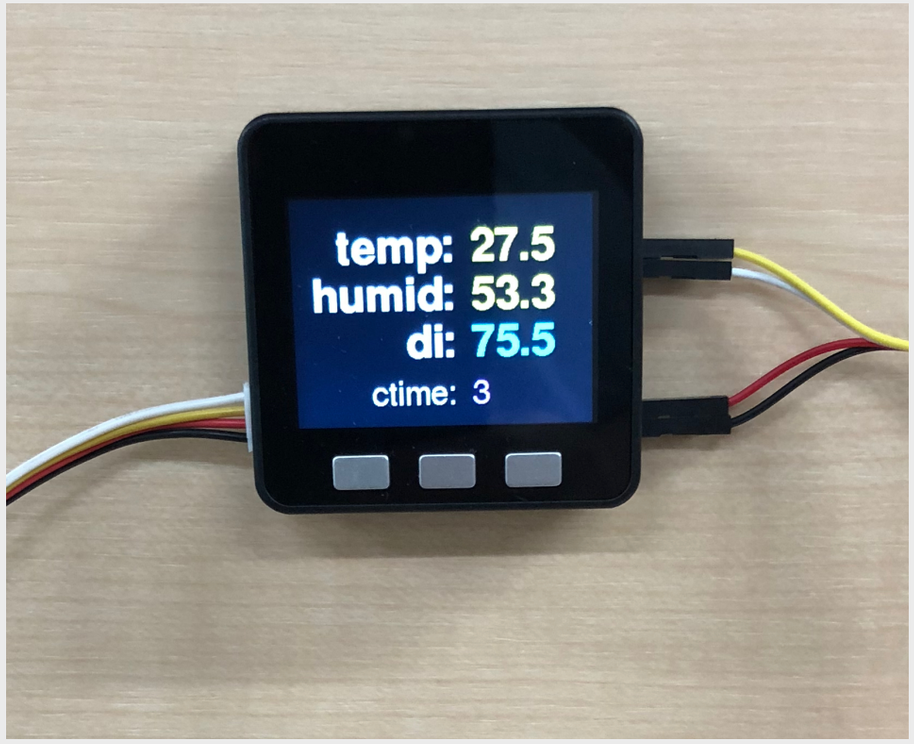
- 時刻と共に連続的，離散的に変化する値の抽象化
  - 例: ボタンの押下，マウスポインタの位置，センサからの入力値

背景：Emfrp [Sawada et al. 2016]

- 小規模組込みシステム向けFRP言語
  - 計算リソースに制限のある組込み環境での動作を想定
    - マイクロコントローラ上(AVR，STM32，ESP32等)での動作
  - **@last**による直前値の参照 (foldpの一般化)
  - 実行時使用メモリの静的な決定



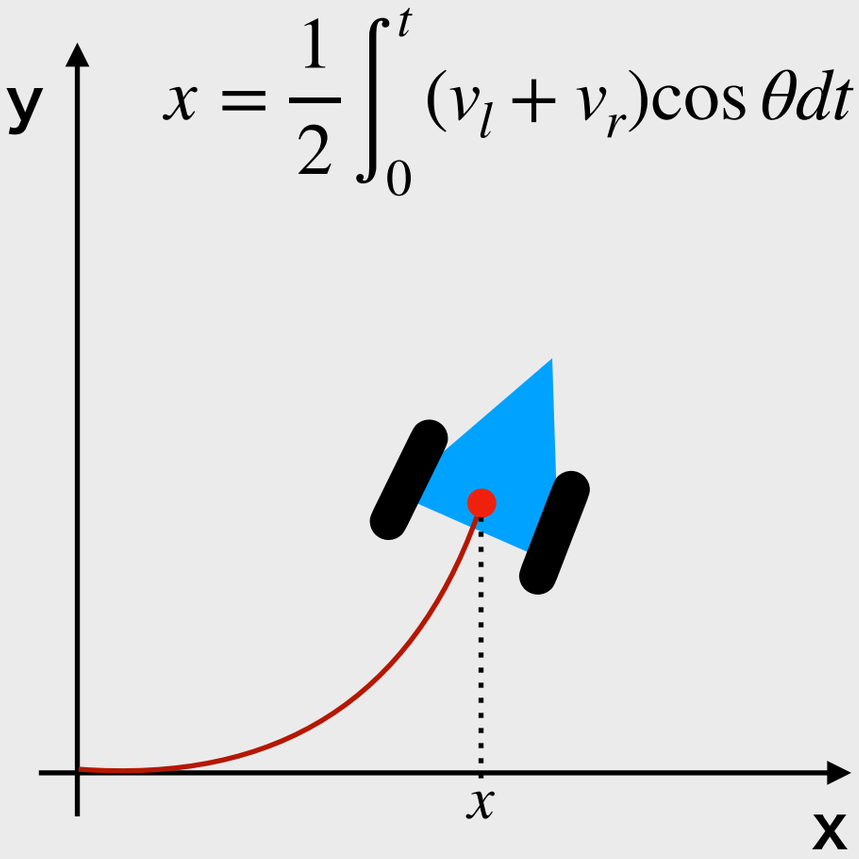
Zumo (CPU: ATmega328p 16MHz, Flash: 32KB, RAM: 2KB)



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

背景：Emfrp [Sawada et al. 2016]

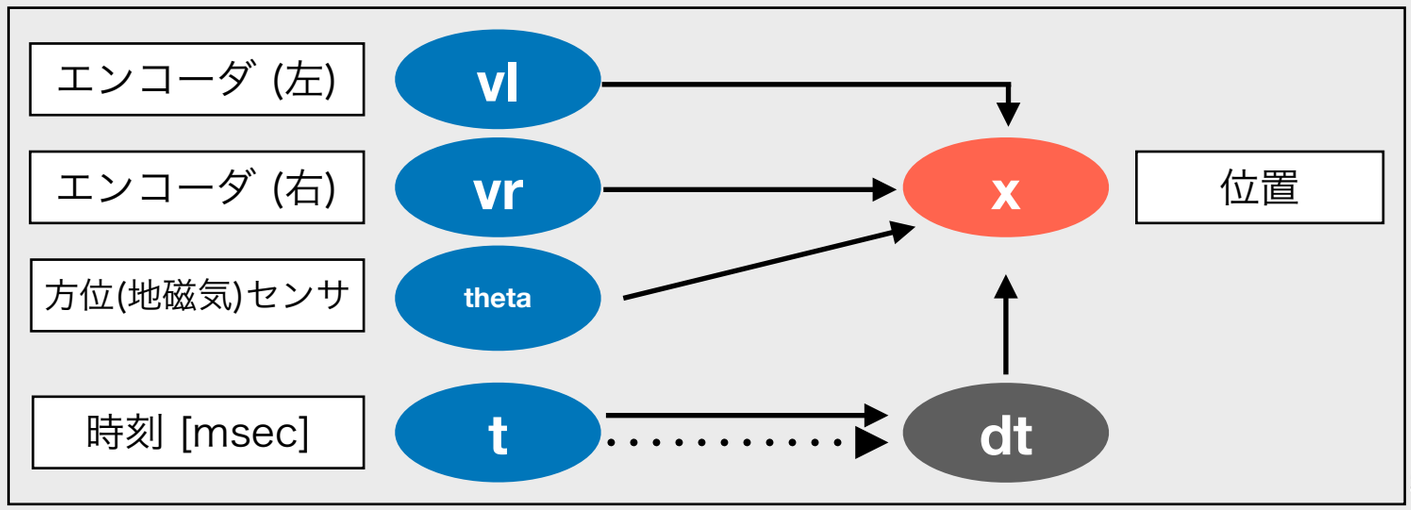
- 小規模組込みシステム向けFRP言語
  - 計算リソースに制限のある組込み環境での動作を想定
    - マイクロコントローラ上(AVR，STM32，ESP32等)での動作
  - **@last**による直前値の参照 (foldpの一般化)
  - 実行時使用メモリの静的な決定



ロボットの位置を計算するEmfrpモジュール

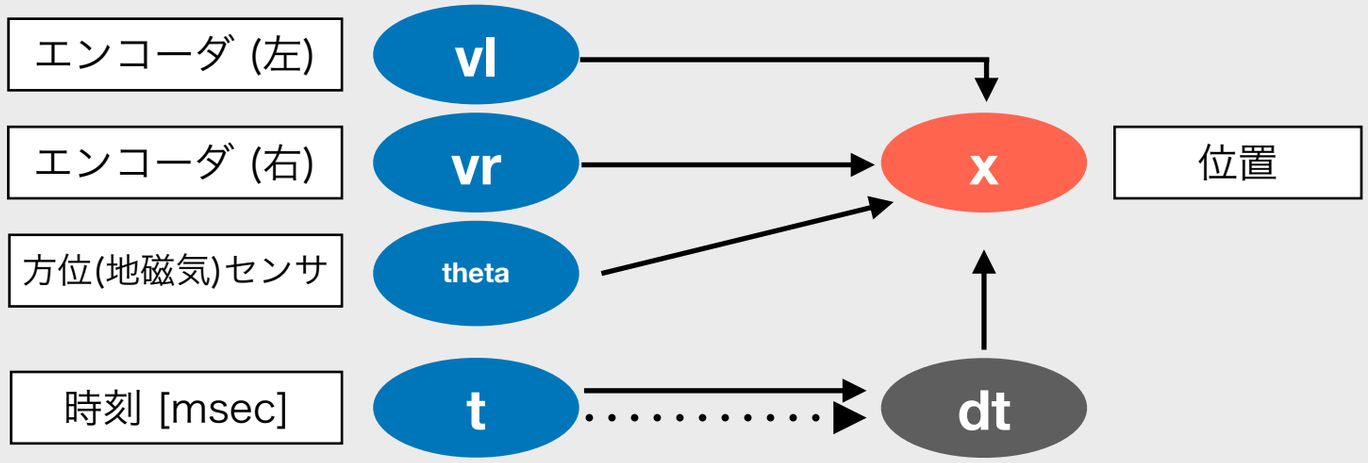
```
module Distance # ロボットの位置（x座標）

in vl : Float, vr : Float, theta : Float, t(0) : Int
out x : Float
use Std

node dt = (t - t@last) / 1000.0
node init[0.0] x = x@last + (vr + vl) * cos(theta) * dt / 2
```

ノード(時変値)の依存グラフ



Emfrpの実行モデル

時変値の依存グラフ



時変値の更新サイクル

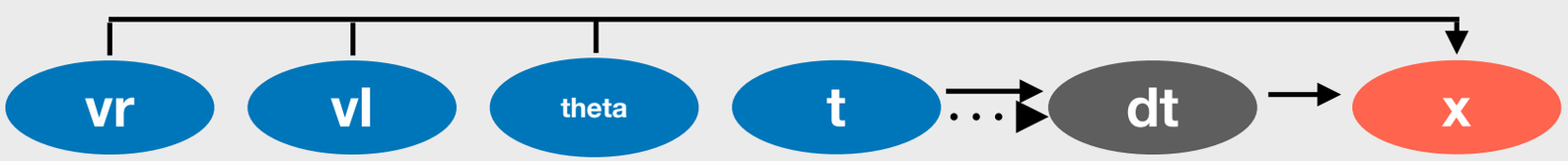
```
// 外部入力関数（C言語）
void Input(
  float* vr, float* vl,
  float* theta, int* t)
{ /* センサから入力 */ }
```

```
// 外部出力関数（C言語）
void Output(float x)
{ /* モニタへ出力 */ }
```

入力関数呼び出し
外部入力を入力ノードに設定

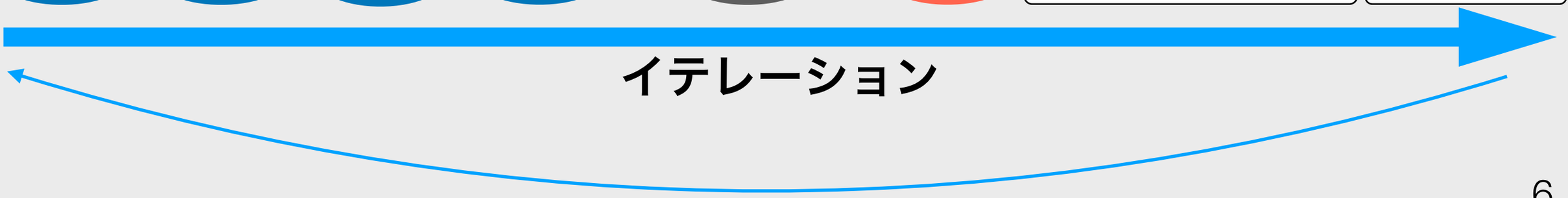
ノード更新処理

出力関数呼び出し
出力ノードを外部へ出力



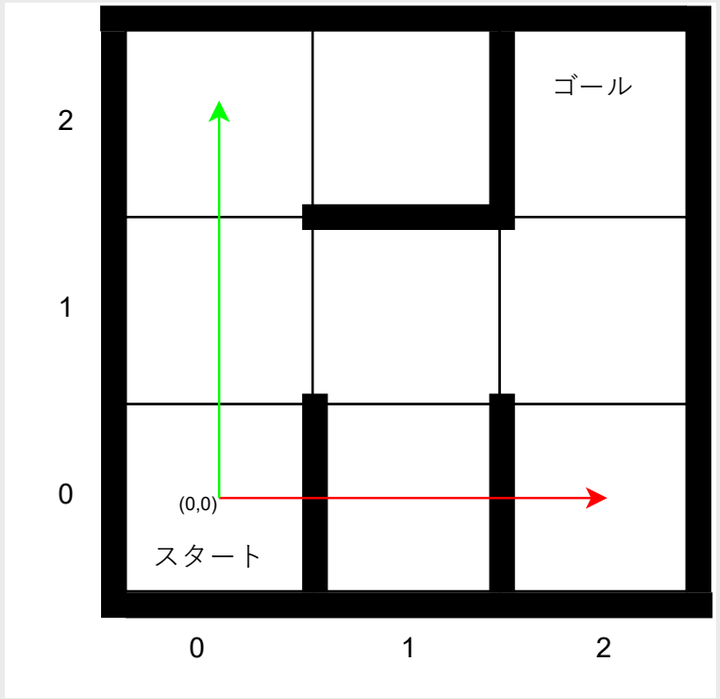
直前値更新
現在値を直前値(@last)へ設定

メモリ管理
不要メモリの解放

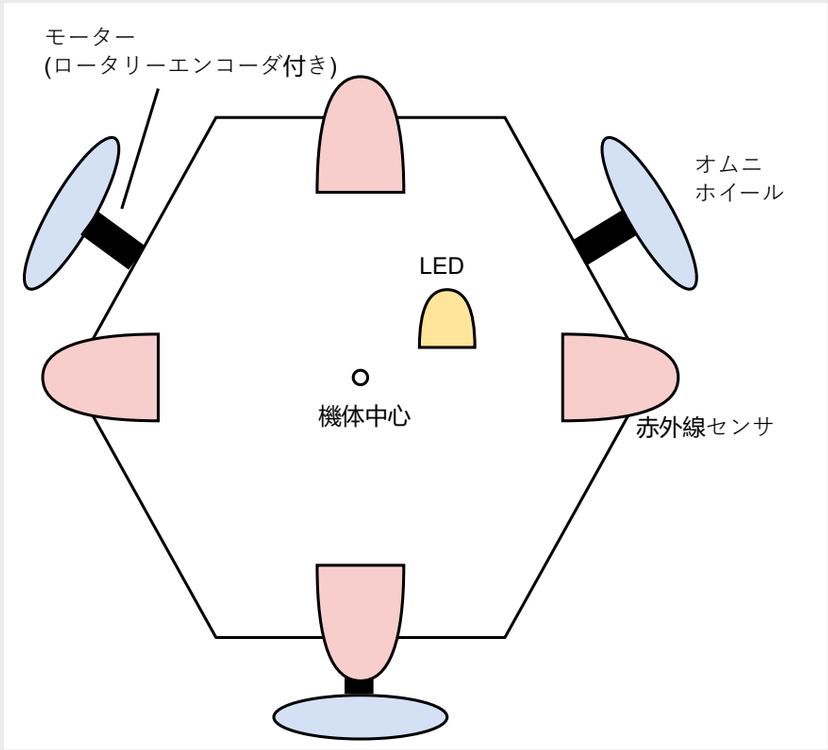


動機：迷路探索ロボットの例

- ロボットが迷路を探索する
  - スタートから出発，自律的に迷路を探索，ゴールしたらLEDを点灯させる
  - ロボット競技「マイクロマウス」の簡略版
- 迷路探索ロボット
  - 全方向移動車
    - オムニホイール，ロータリーエンコーダ(回転計)あり
  - 東西南北の壁を検知するセンサー(赤外線)



迷路の例



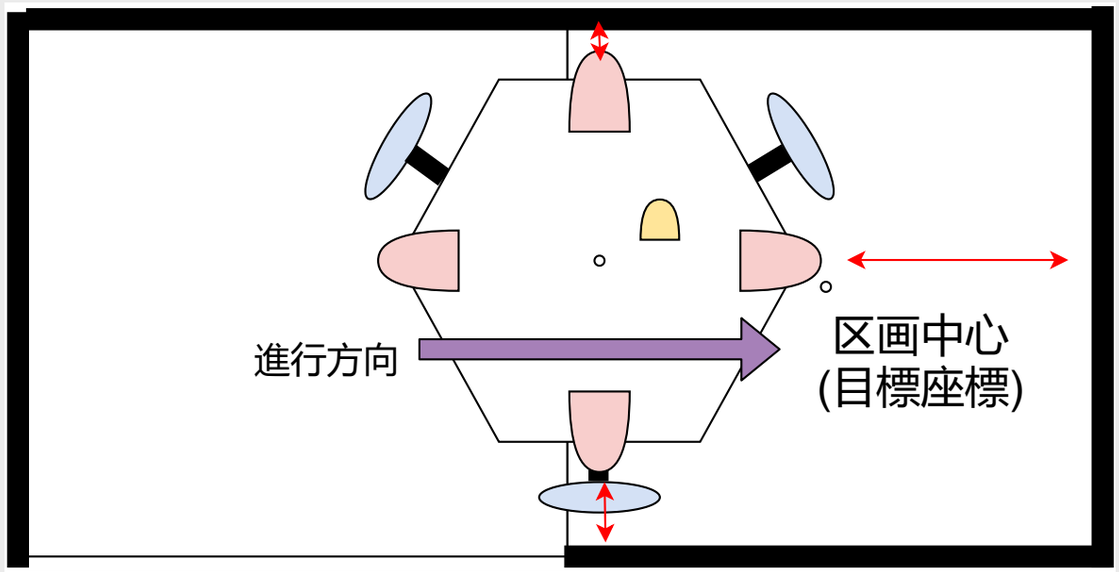
ロボット概要図

オムニホイール
(引用:Wikipedia)

動機：迷路探索ロボットの例

- ロボットの動作 (以下をゴールするまで繰り返す)
  - A: 目標座標へ移動 (リアクティブな動作)
  - B: 壁センサの情報で「迷路」を更新
  - C:「迷路」を探索して次の目標座標を計算

- 動作の改善
  - 目標座標へ到着する前に，壁情報の更新が可能
    → Aと(B，C)を並行に実行できれば「計算」待ちが軽減



動作改善が行える状況

Heavy-taskとEmfrpの機能制限

Heavy-task

- **イテレーション毎の実行が必ずしも必要ではないが，比較的時間がかかる処理**
  - イテレーション中に実行されると応答性が悪化する
- ある程度複雑なデータ構造の操作処理を想定
  - 前述例では，「迷路」の更新処理，探索処理

タスクリソース

- Heavy-taskが操作の対象とするオブジェクト
  - 前述例では，「迷路」を表すデータ構造

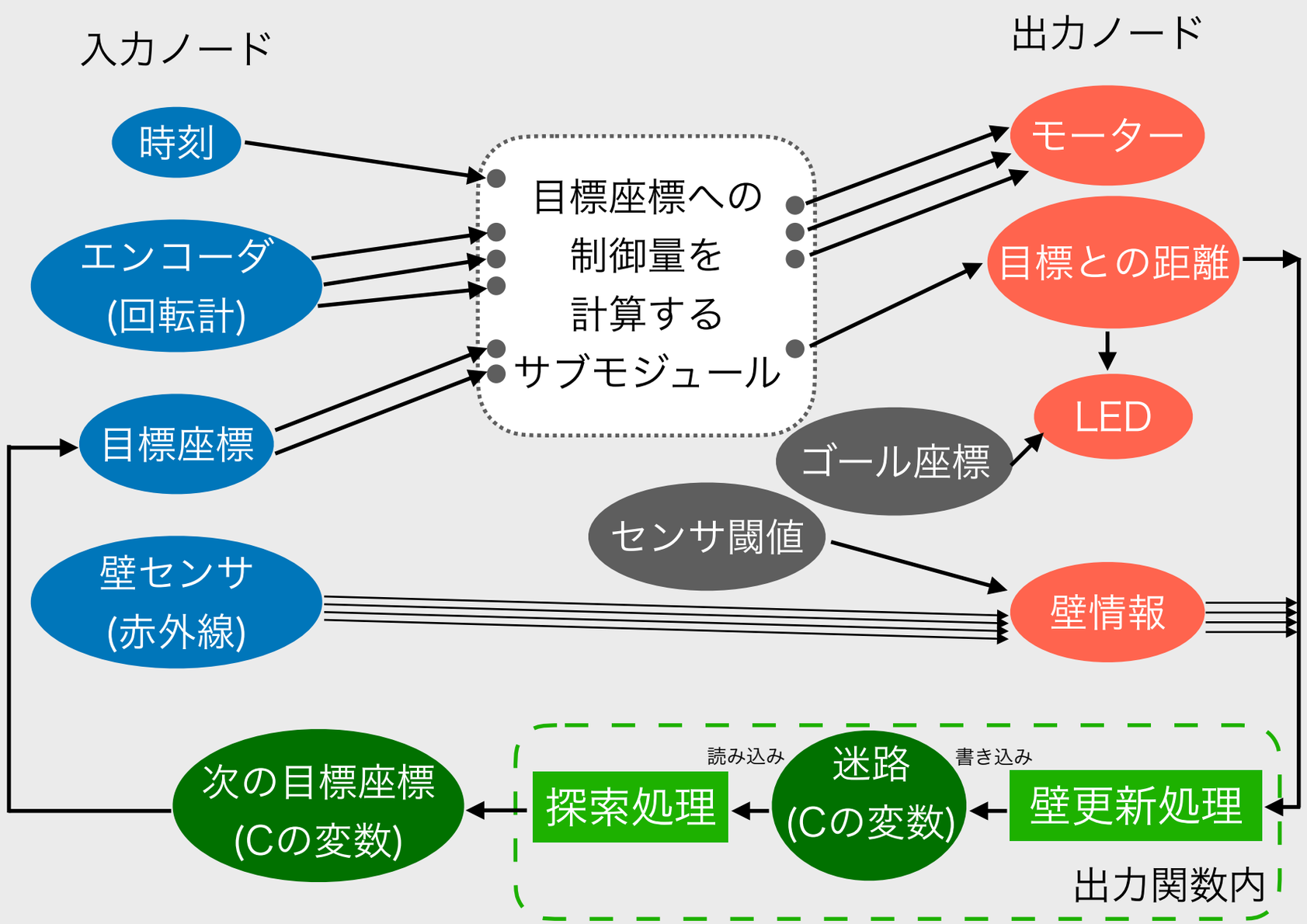
Emfrpの機能制限

- 時変値が第一級ではない → 時間/空間漏れの防止
- 再帰データ型，再帰関数の禁止 → 実行時使用メモリ決定
  - 前述例での「迷路」やその更新，探索処理が記述できない
    → **外部関数呼び出し (C言語による記述)**

従来のEmfrpによる迷路探索ロボット設計

出力から入力へのフィードバックによるHeavy-task実行

ノード間の依存関係のグラフ



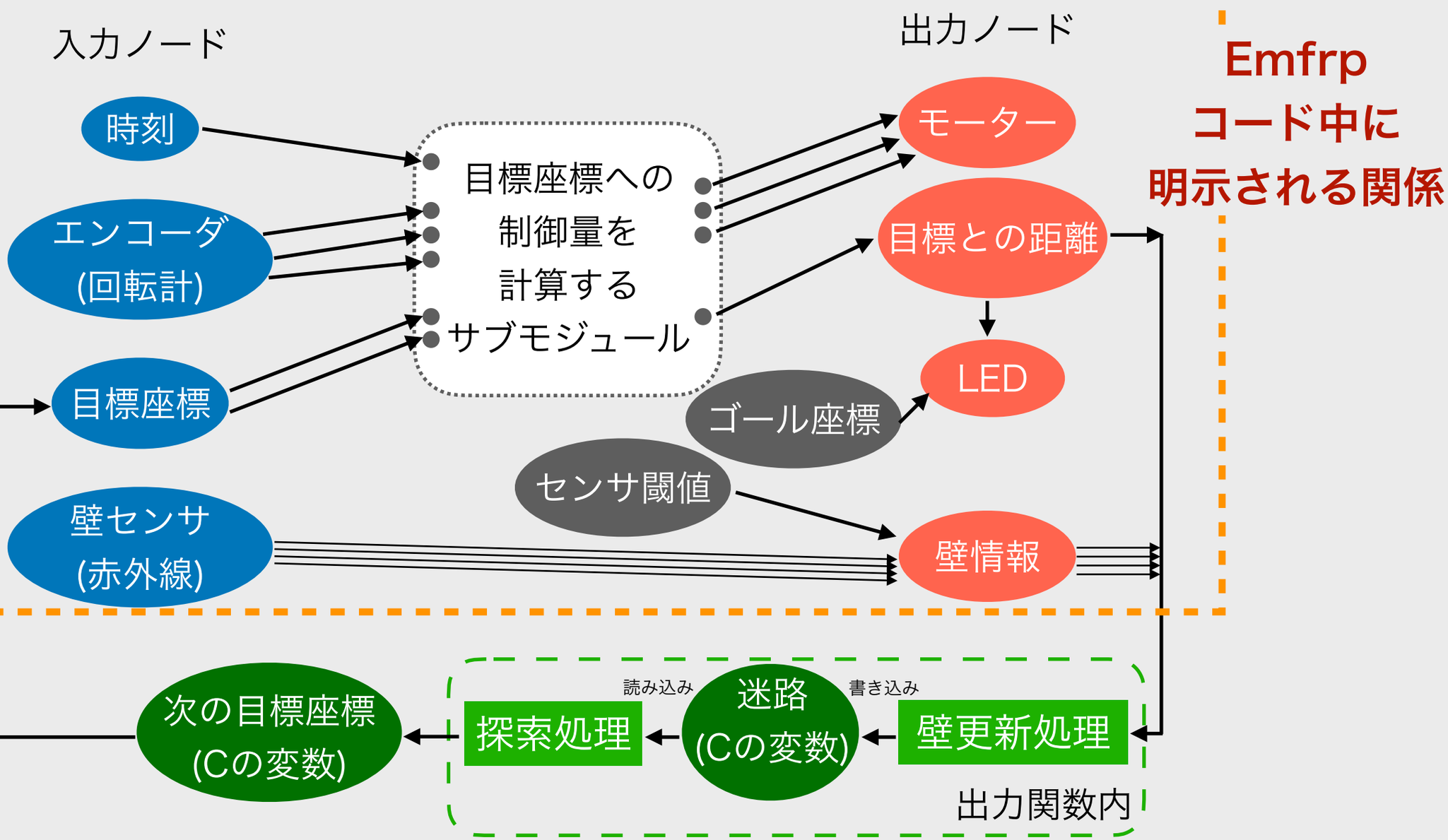
問題点

- ① リアクティブな処理とHeavy-taskの並行実行はできない
  - ノード更新処理の**応答性が悪くなる**ことがある

従来のEmfrpによる迷路探索ロボット設計

出力から入力へのフィードバックによるHeavy-task実行

ノード間の依存関係のグラフ



Emfrp コード中に明示される関係

問題点

- ② 出力から入力への**暗黙的な依存関係**
  - Emfrpコード中に「次の目標座標の計算」が現れない

従来のEmfrpによる迷路探索ロボット設計

出力から入力へのフィードバックによるHeavy-task実行

- RTOS等の並行実行基盤との協調
  → リアクティブな処理とHeavy-taskの並行実行が**可能**
  - Emfrpランタイムとの適切な協調
  - **タスクリソース**(「迷路」を表すデータ構造)の排他制御が必要

問題点

- ② 出力から入力への**暗黙的な依存関係**
- ③ リソースの適切な排他制御
  - 従来の**並行プログラミングの難しさ**が顕在化
    → FRPを利用する利点が損なわれる

提案手法：言語拡張

- 問題点①②③を解決するために以下を導入
  - タスクリソースとそれを扱うHeavy-taskの定義
  - tasknode定義によるHeavy-task発行機構
  - Heavy-task実行の計算状態を表すFuture型

提案手法：言語拡張

- 問題点①②③を解決するために以下を導入
  - **タスクリソースとそれを扱うHeavy-taskの定義**
  - tasknode定義によるHeavy-task発行機構
  - Heavy-task実行の計算状態を表すFuture型

タスクリソースを定義
(実体はC構造体)

Heavy-taskの名前と型を定義
(実体はC関数)

排他制御のための注釈

```
resource MazeGraph { #  「迷路」を表すデータ構造の抽象化
  # 壁情報をMazeGraphのインスタンスに記録するタスク
  RegisterSection : (u : Int, v : Int, n : Bool, …… ) -> (h : Unit) / write

  # 次に行くべき区画(next_u,next_v)を計算するタスク
  CalcNextSection :
    (u : Int, v : Int, goal_u : Int, goal_v : Int) -> (next_u : Int, next_v : Int) / read
}
```

```
module MazeRunner
in mg : resource MazeGraph, # heavy-taskリソースの指定
   sn : Int, se : Int, ss : Int, sw : Int, # 赤外線センサー
   ……
out achieved : Bool, # ゴールに到着したかどうか
   ……
```

タスクリソースのインスタンスは
プログラム開始時に外部から
モジュールへ渡される

提案手法：言語拡張

- 問題点①②③を解決するために以下を導入
  - タスクリソースとそれを扱うHeavy-taskの定義
  - **tasknode定義によるHeavy-task発行機構**
  - Heavy-task実行の計算状態を表すFuture型

Future型のノード定義

```
# 区画情報の登録
tasknode finish_register : Future[Unit] =
  mg.RegisterSection(to_u, to_v, wall_n, wall_e, wall_s, wall_w)
    trigger (!in_target_section@last && in_target_section)
                # 区画の中心にある程度近づいた瞬間(立ち上がりエッジ)
```

扱うタスクリソースの指定と
発行するタスクの指定

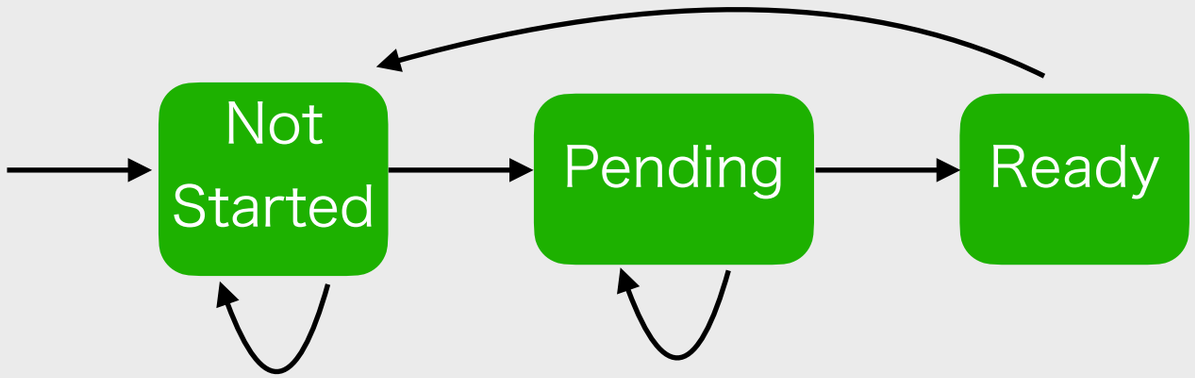
タスクの発行条件

提案手法：言語拡張

- 問題点①②③を解決するために以下を導入
  - タスクリソースとそれを扱うHeavy-taskの定義
  - tasknode定義によるHeavy-task発行機構
  - **Heavy-task実行の計算状態を表すFuture型**

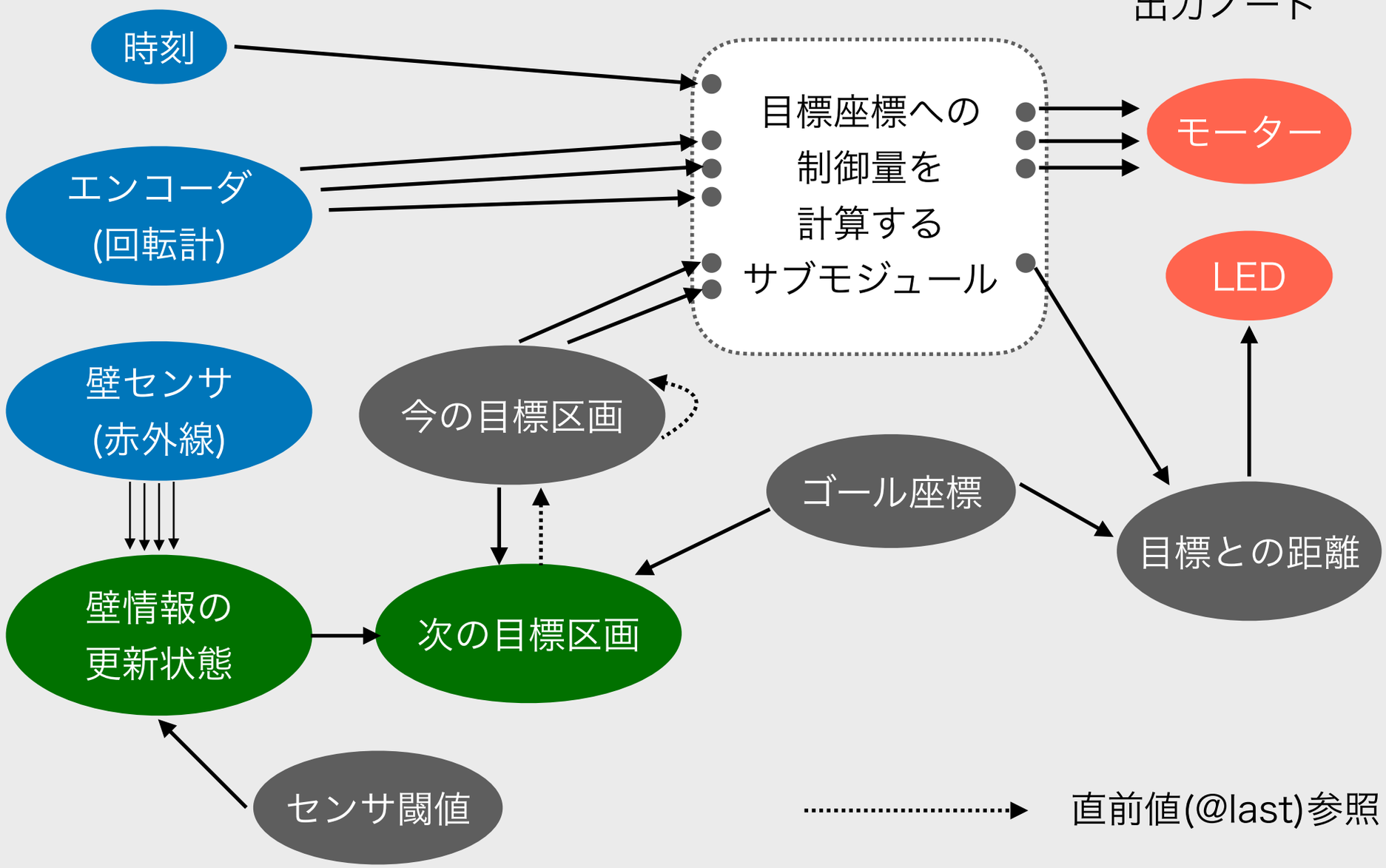
```
# 結果がA型のHeavy-taskの計算状態を表す型
type Future[A] =
  | NotStarted # タスクが発行されていない
  | Pending    # タスクの計算終了待ち
  | Ready(A)   # タスクの実行結果(計算終了後1イテレーションのみ)
```

Future型の状態遷移



拡張Emfrpによる迷路探索ロボット設計

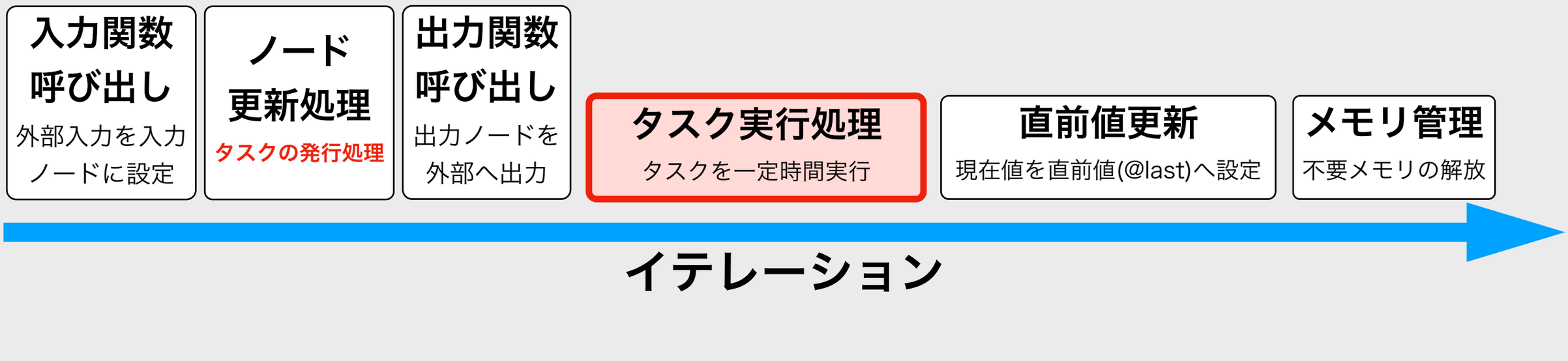
ノード間の依存関係のグラフ(一部省略)



- 出力から入力への暗黙的な依存関係が発生しない
- **tasknode**ノード(緑色)は通常ノード更新と並行に非同期実行される

提案手法：ランタイム拡張

- タスク実行を行うフェーズを追加



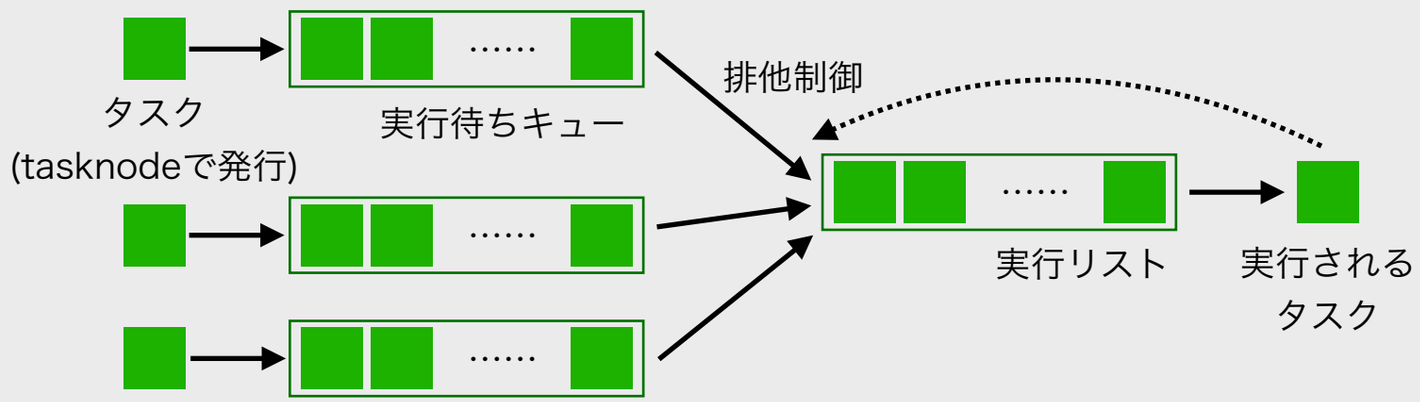
- タスクの分割実行
  - **プリエンプティブ**
    - タイマ割り込み
    - 最長連続実行時間はコンパイル時に指定
  - 一般的な**コンテキストスイッチ**技法

提案手法：ランタイム拡張

- タスク実行のスケジューリング
  - **ラウンドロビン方式**
    - 1イテレーションにつき1タスク実行

- シンプルなタスク実行管理アルゴリズム
  - シングルコアマイコン上での動作を想定
  - タスクリソースの排他制御
    - `resource`定義のタスクに対する注釈(`read`/`write`)を利用
    - 実行リスト(ランタイムに1つ)，実行待ちキュー(リソース毎に用意)

タスク管理アルゴリズム概要図



関連研究

- futureパターン
  - 並行プログラミングにおけるデザインパターン
  - 非同期処理の結果の取得を遅延させる
    - Heavy-taskは非同期処理されることが望ましい
    - Heavy-taskの計算状態管理が必要

- Actor-Reactorモデル[Van den Vonder et al. 2020]
- Actor(副作用のある処理やHeavy-taskを行う)とReactor(リアクティブな動作を行う)の分離
  - ActorとReactorの協調
    - Reactorの出力からActorを経由し，Reactorの入力へとフィードバック
    - 暗黙的な依存関係

まとめ

- 小規模組込みシステム向けFRP言語Emfrpに対して，十分な応答性を保ちながらHeavy-task実行を可能にする非同期タスク処理機構の導入を提案した
- 具体例の説明を通して，その機構の有用性を示した

今後の展望

- 提案した機構を組み込んだ処理系の実装
  - 時間的，空間的オーバーヘッドの評価
- 意味論の形式化
  - リアクティブな挙動とHeavy-task実行の協調動作に関する諸性質の証明