

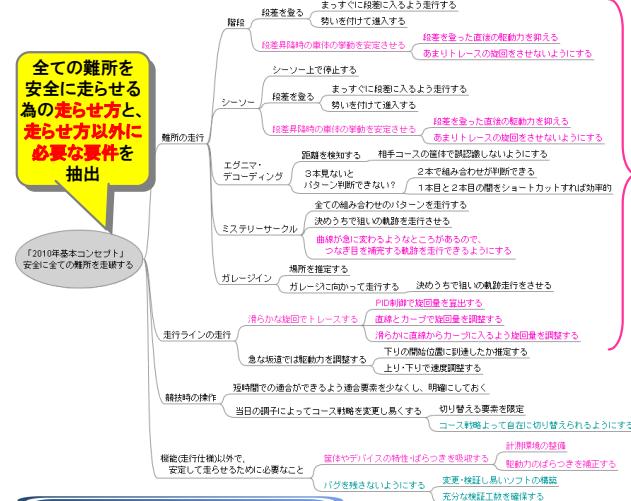
1. 要件分析

安全に走行するために必要な要件を抽出する

■目標(コンセプト): コース全域(難所を含む)を安全に走破する

[I] コンセプト実現の為の要件分析

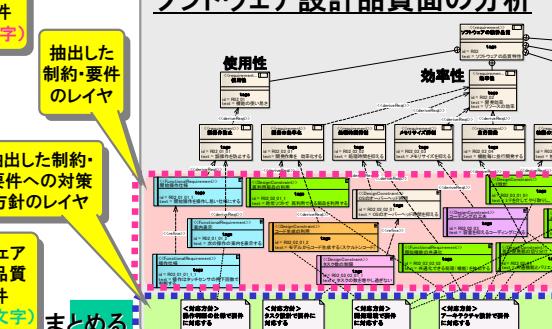
※ コンセプトからコース全域の安全な(=安定した)走行に必要な要件を抽出



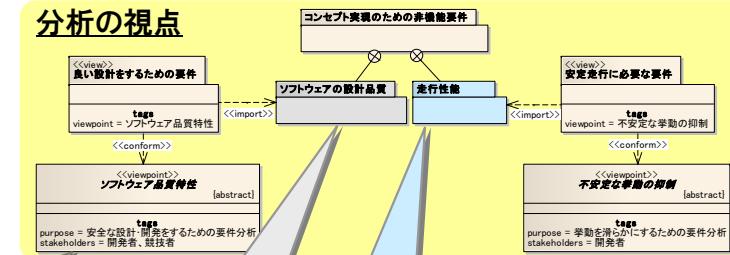
[II] 非機能要件の分析

※ [I] の要件分析より、安定して走行させる為の非機能要件を「走行性能の要件」と「ソフトウェア設計品質の要件」の両面からSysMLを使って分析

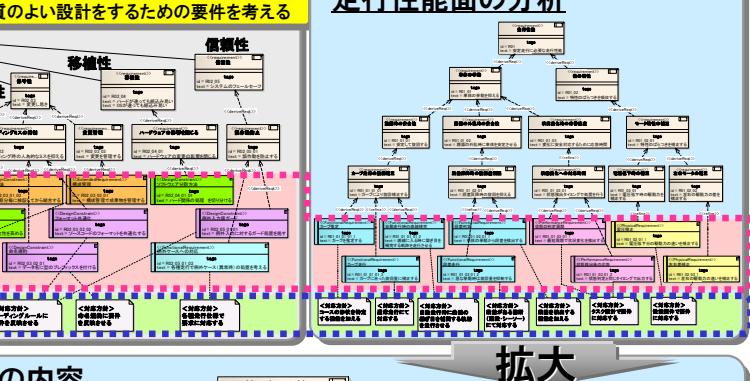
ソフトウェア設計品質面の分析



分析の視点



走行性能面の分析



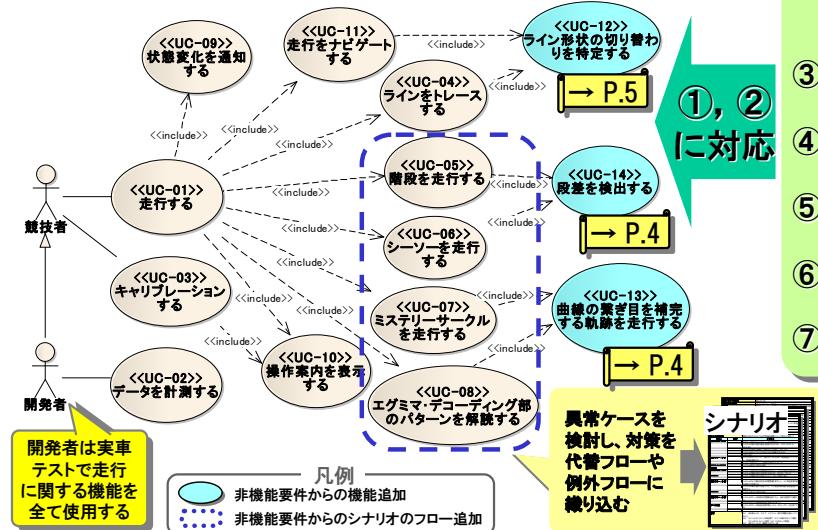
<制約・要件への対応方針>

- ① 機能追加 → P.1
- ② シナリオの代替・例外フローの検討 → P.1
- ③ タスクの設計 → P.3
- ④ アーキテクチャ設計 → P.2
- ⑤ コーディング規約 → P.5
- ⑥ 命名規約 → P.5
- ⑦ 開発環境 → P.5

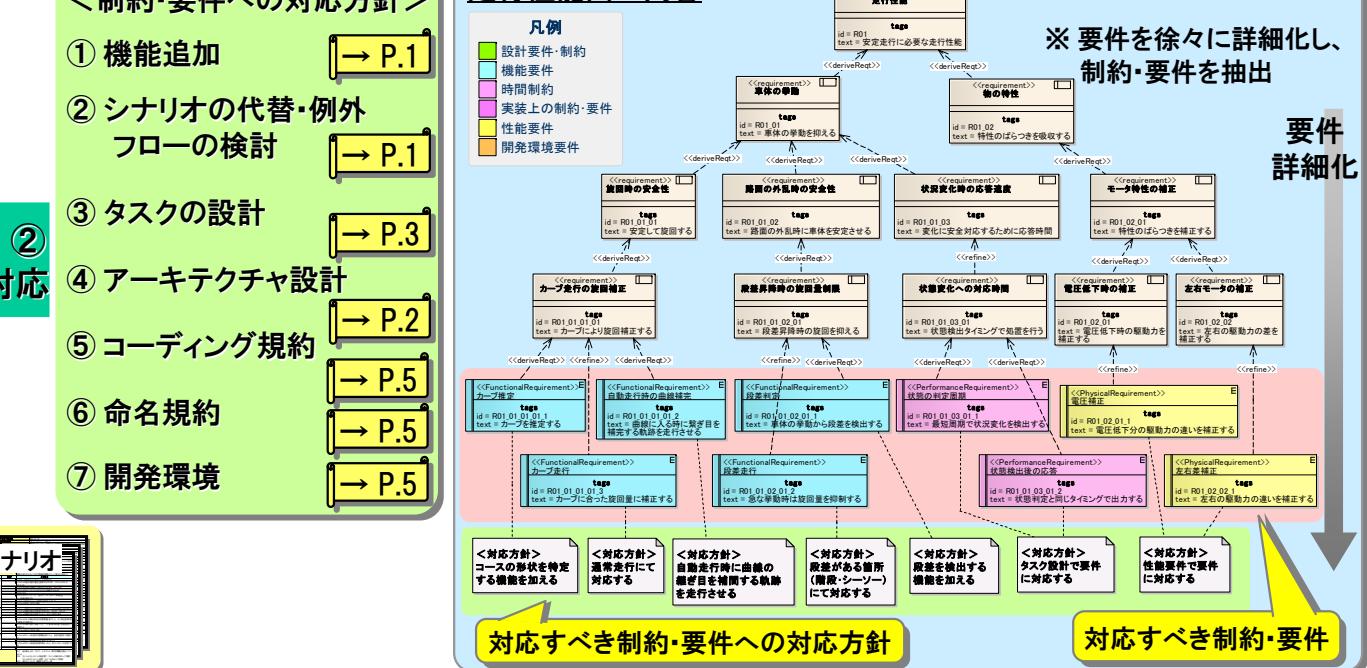
①, ②
に対応

[III] 機能要件

※ [I] の要件分析で抽出した機能要件に、非機能要件の分析結果から機能要件に関するものに反映する



走行性能面の内容



※ 要件を徐々に詳細化し、制約・要件を抽出

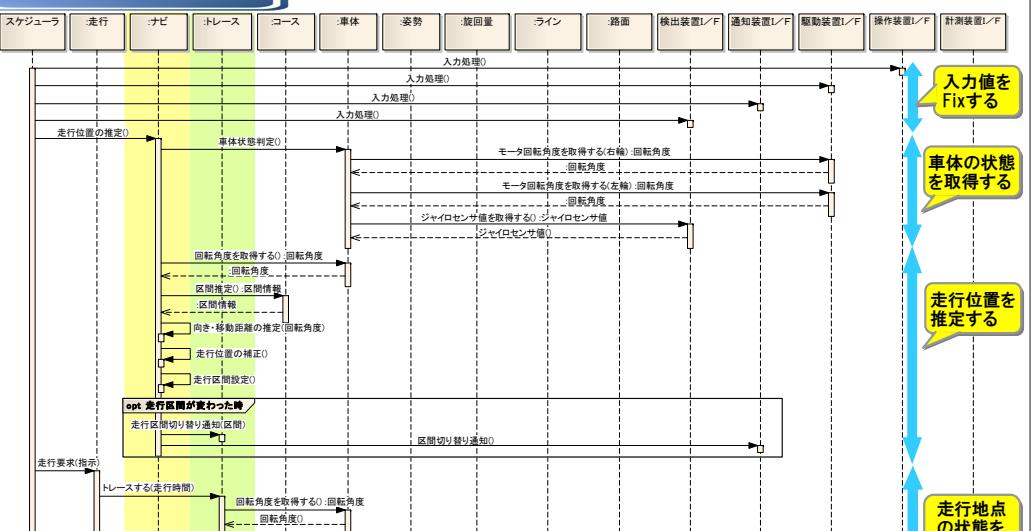
要件
詳細化

3. 処理の流れ

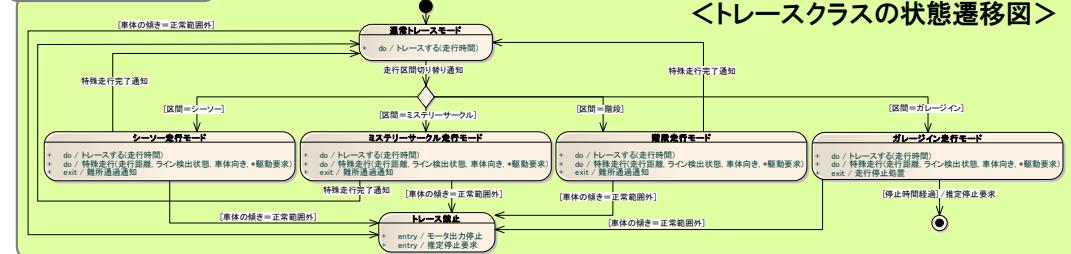
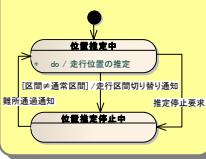
全体の処理の流れとタスク設計

[I] 全体の流れ

例) 主要部分(ETロボコンシステムブロック)の通常走行時のシーケンス



<ナビクラスの状態遷移図>



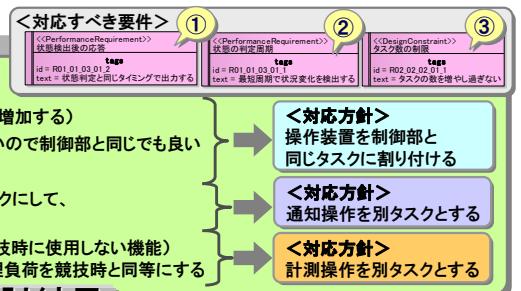
[II] タスク設計と並行性

タスク分割 (SysML: ブロック図)

<タスク分割設計時の検討事項> 検討要件に反映

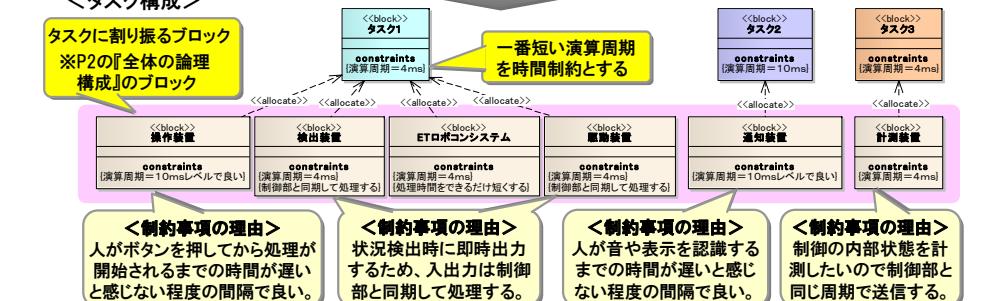
- ① タスクの数を増やしすぎたくない (OSのオーバーヘッド時間が増えるため処理時間が増加する)
- ② 操作装置は走行前の処理で走行中の処理負荷はないので制御部と同じでも良い
- ③ 同期して実行させたい処理は同じタスクに割り付ける
- ④ 時間がかかる又は長い演算周期で良い処理は別タスクにして、制御部の処理負荷を軽減する
- ⑤ 本来の機能以外の負荷機能(開発時だけ使用して競技時に使用しない機能)を別タスクにして、負荷機能使用時にも制御部の処理負荷を競技時と同等にする

* 論理部品(ブロック)のタスクへの割り付けを検討し、各タスクの時間制約を検討する



検討結果

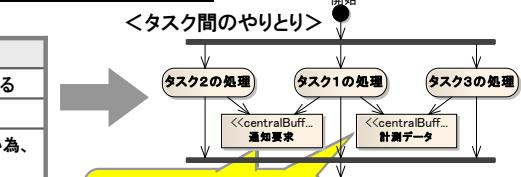
<タスク構成> タスクに割り振るブロック ※P2の「全体の論理構成」のブロック



タスクの優先順 ※タスクの責務と時間制約から優先度を決める

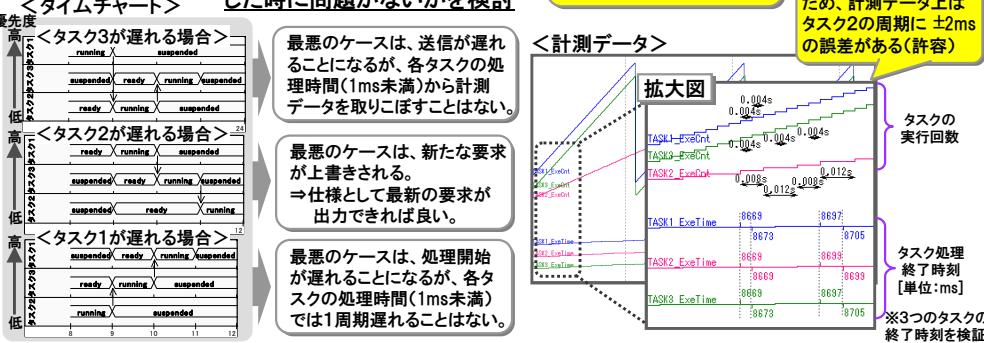
<優先順一覧> ※優先度: 1(高) → 3(低)

タスク名	優先度	周期	理由
タスク1	1	4ms	制御部は必ず4ms毎に実施する
タスク2	3	10ms	少し(数ms)遅れても問題ない
タスク3	2	4ms	計測データを取りこぼさない為、制御部と同じ周期で処理する



並行性の検証 ※タスクが重なって遅れが発生した時に問題がないかを検討

<タイムチャート>



5. 性能

周回タイムの短縮化／非機能要件の実現／開発環境

[I] 自己位置推定 & PID制御

「PID制御」と「最適な旋回要求指示」を複合させ、コース全域で安定したライントレースを実現！

安定したライントレースのためのコンセプト

課題

- 直線を滑らかに走れるようにPID制御を調整すると、直線⇒カーブの変移点や曲率が異なるカーブが連続するときに追従できない
- ある曲率のカーブを滑らかに走れるようにPID制御を調整すると、曲率の異なるカーブや直線で車体がふらつく

解決方法

カーブに入ったらPID制御の基準値を旋回半径に合った基準値に補正して、コース全体を滑らかにトレースする！

自己位置推定(ナビ)

<機能概要>

コース全体をカーブの旋回半径が変わる単位で細かく区切り、車体の向きと移動距離から位置(走行中の区間)を特定する。また、走行中の区間の情報(形状・旋回半径)を管理する。

※ 昨年の20分割から60分割に変更してより細かく制御量の切替ができるようになり、ナビ機能も大幅にパワーアップ

<推定方法>

- モータの回転角度から移動距離と車体の向きを把握
 - 区間情報と車体の向き・移動距離を照合して走行位置を推定する
- ※ 区間の区切りは図5-1を参照

PID制御の基準値の補正方法

<機能概要>

カーブ走行時に、光センサ値の走行閾値との差によるPID制御の基準値を走行中の区間の旋回半径から算出した旋回量(最適旋回量)に補正することで、カーブの走行を滑らかにする。(図5-2参照)

<補正方法>

- モータ回転角度から移動距離、車体旋回角度を求め、式5-1から走行区間の旋回半径上の軌跡を取る旋回量(最適旋回量)を求める(図5-3参照)
- 旋回量(最適旋回量)を求める際に必要となるモータ駆動力は、倒立制御で調整されるため、最終的に出力するモータ駆動力をフィードバックさせる(図5-4参照)
- 光センサ値の走行閾値との差から算出したPID制御量に旋回半径から算出した旋回量(最適旋回量)を加算する

試走会での検証の結果、コースの摩擦の違いによるズレは、PID制御で吸収できる範囲内 ⇒60分割したコース情報の適合は不要

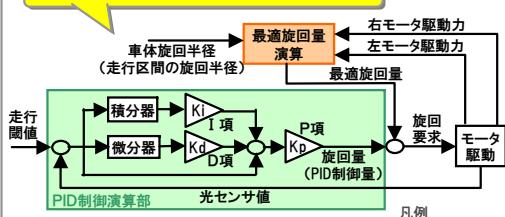


図5-4. 旋回要求指示ブロック図

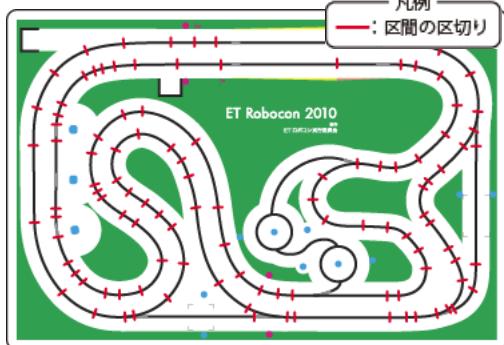


図5-1. 区間分割(ナビ情報)

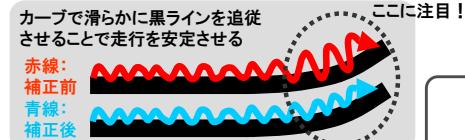


図5-2. ライントレースのイメージ

<検証結果>
アウトコースを1周走行時のPID旋回量の波形

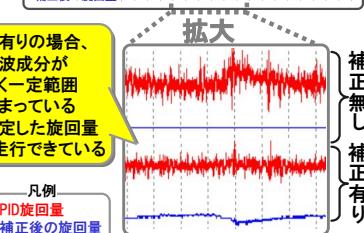
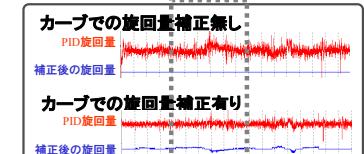


図5-5. 旋回要求指示式

[II] 構造解析

UMLモデルとソースコードをDSMIにより可視化して依存関係を管理することで、ソフトウェアの品質を確保！

ソフトウェアの設計構造(アーキテクチャ)解析

非機能要件の一つとなるアーキテクチャ設計について、ソフトウェア品質特性(ISO/IEC9126)のうち、「保守性」「移植性」を維持する為の設計ルールが守られているかを市販ツール(Lattix/Understand)により確認。ソフトウェアの関連が複雑化するような変更をしていないか検査し、防止することで上記の品質を維持。

モデルの解析結果

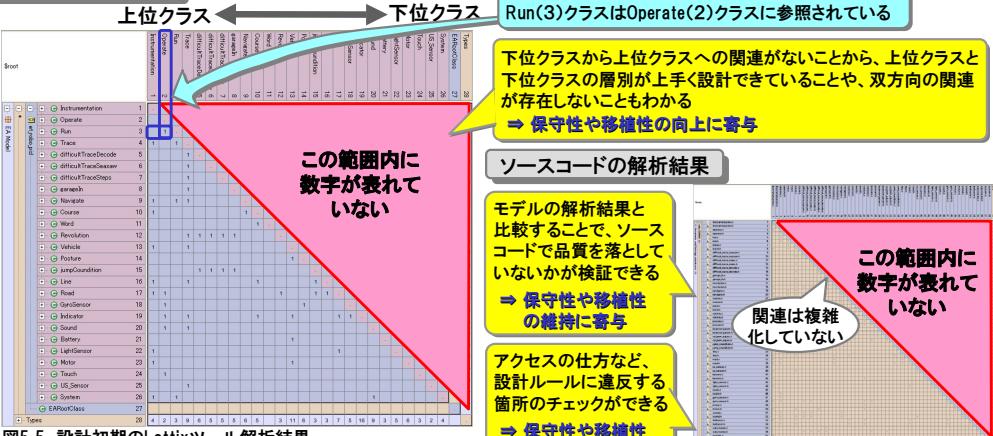


図5-5. 設計初期のLattixツール解析結果

△図5-4の読み方
Run(3)クラスはOperate(2)クラスに参照されている

下位クラスから上位クラスへの関連がないことから、上位クラスと下位クラスの層別が上手く設計できていることや、双方向の関連が存在しないこともわかる
⇒ 保守性や移植性の向上に寄与

ソースコードの解析結果

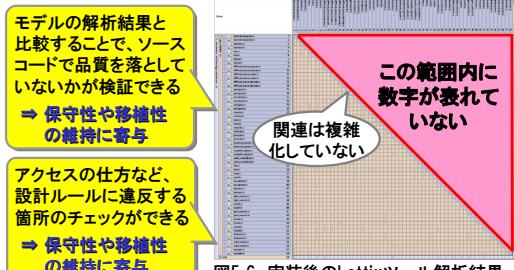


図5-6. 実装後のLattixツール解析結果

[III] 開発環境

ソフトウェアの安全設計のために、開発成果物を管理し、ミスを抑制するためのルールと特性や性能を検証するための計測環境を整備。

成果物管理

モデル ソースコード
単体テスト環境のプロジェクト

登録
構成管理ツール(VSS)
構成管理ツールにてバージョンを管理する

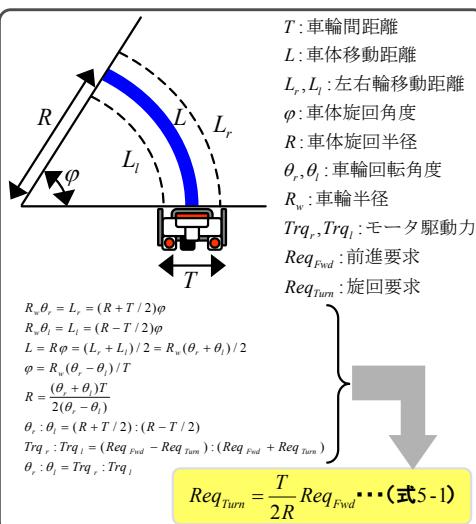
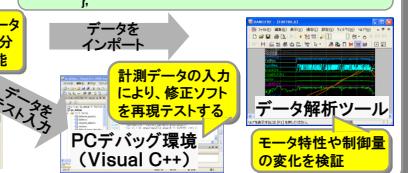


図5-3. 自己位置推定式

△変数の命名の例
型のプレフィックスを付け“_”で区切る
(例) unsigned char u1_var_a;

△条件判定(IF文)の例
IF文は条件が見易いように条件文の頭を揃える
(例) if (u1_wk1 == 0 && u1_wk2 >= 10) { u1_wk3++; }
O if ((u1_wk1 == 0)
&& (u1_wk2 >= 10)) {
u1_wk3++;
};



データをインポート
データ解析ツール
データ割り付けの自由設定機能追加
データを再現テストする
PCデバッグ環境(Visual C++)
モータ特性と制御量の変化を検証