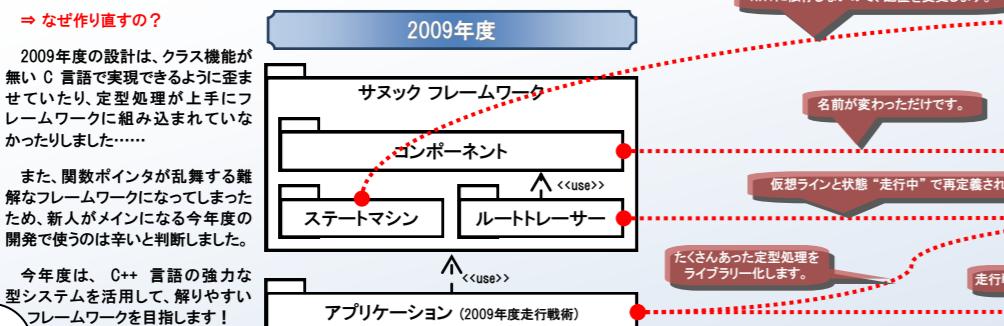


ここでは、SysMLに含まれる『要件図 (Requirement Diagram: 要求図と訳されること)』を利用してやりたいこと、やるべきことを分析しています。
分析した結果、大別して3種類の要件グループになりました。

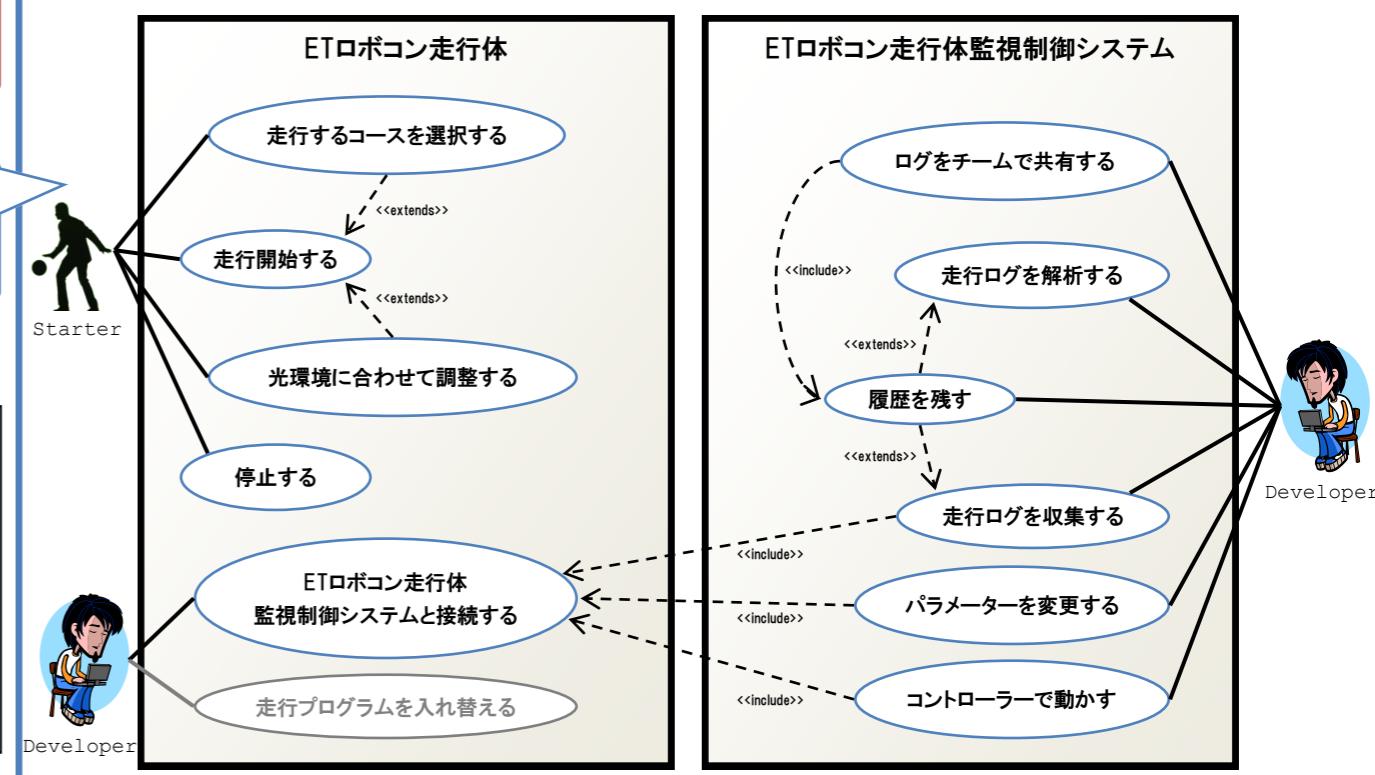
- ① 走行の仕組みに関する要件
- ② 走行戦術に関する要件
- ③ 分析/調整に関する要件

毎度おなじみのユースケース図です。
ユースケース図ではユーザーが行う操作の視点からシステムを分析するため、自律システムを作成するETロボコンにはちょっとミスマッチかもしれません（最初は“走行する”とか“難所を攻略する”とかをユースケースアイコンで書いてしまったものです）。

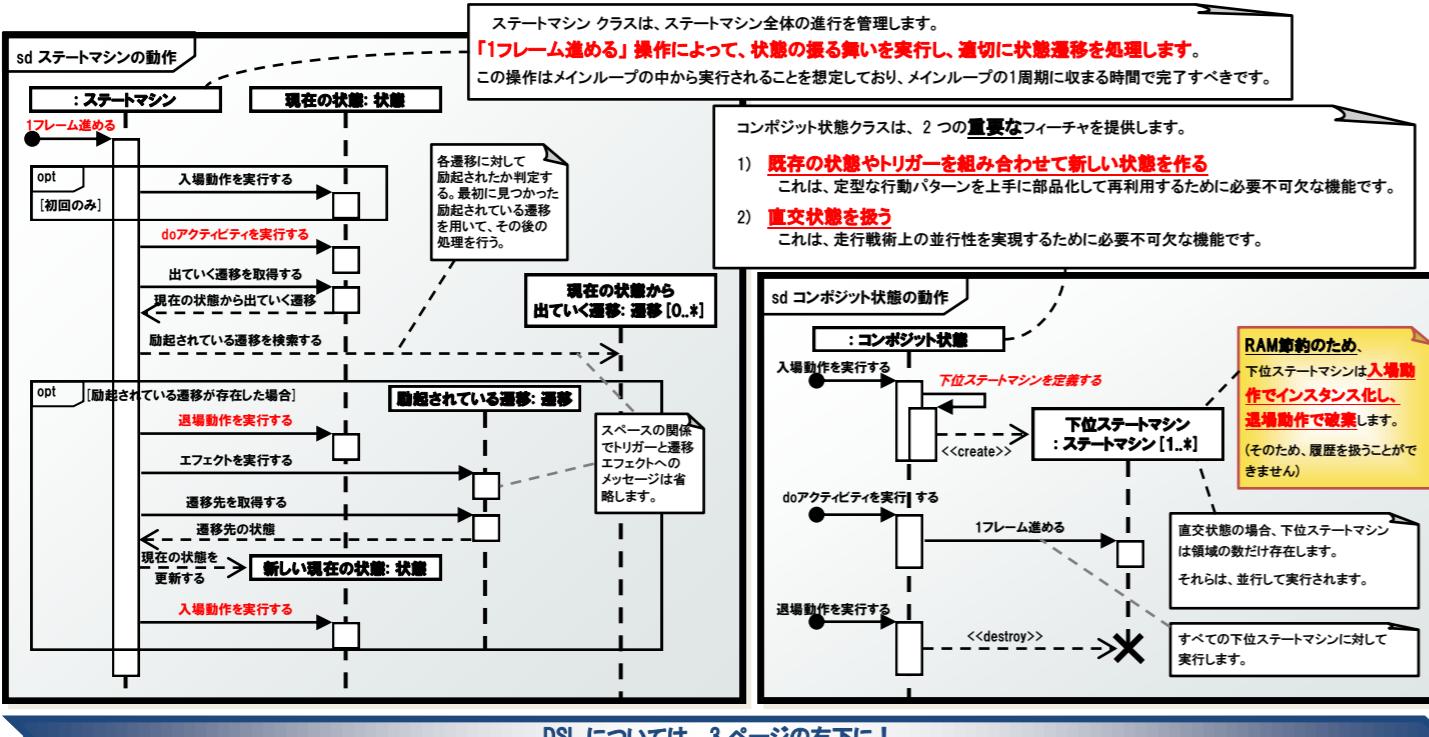
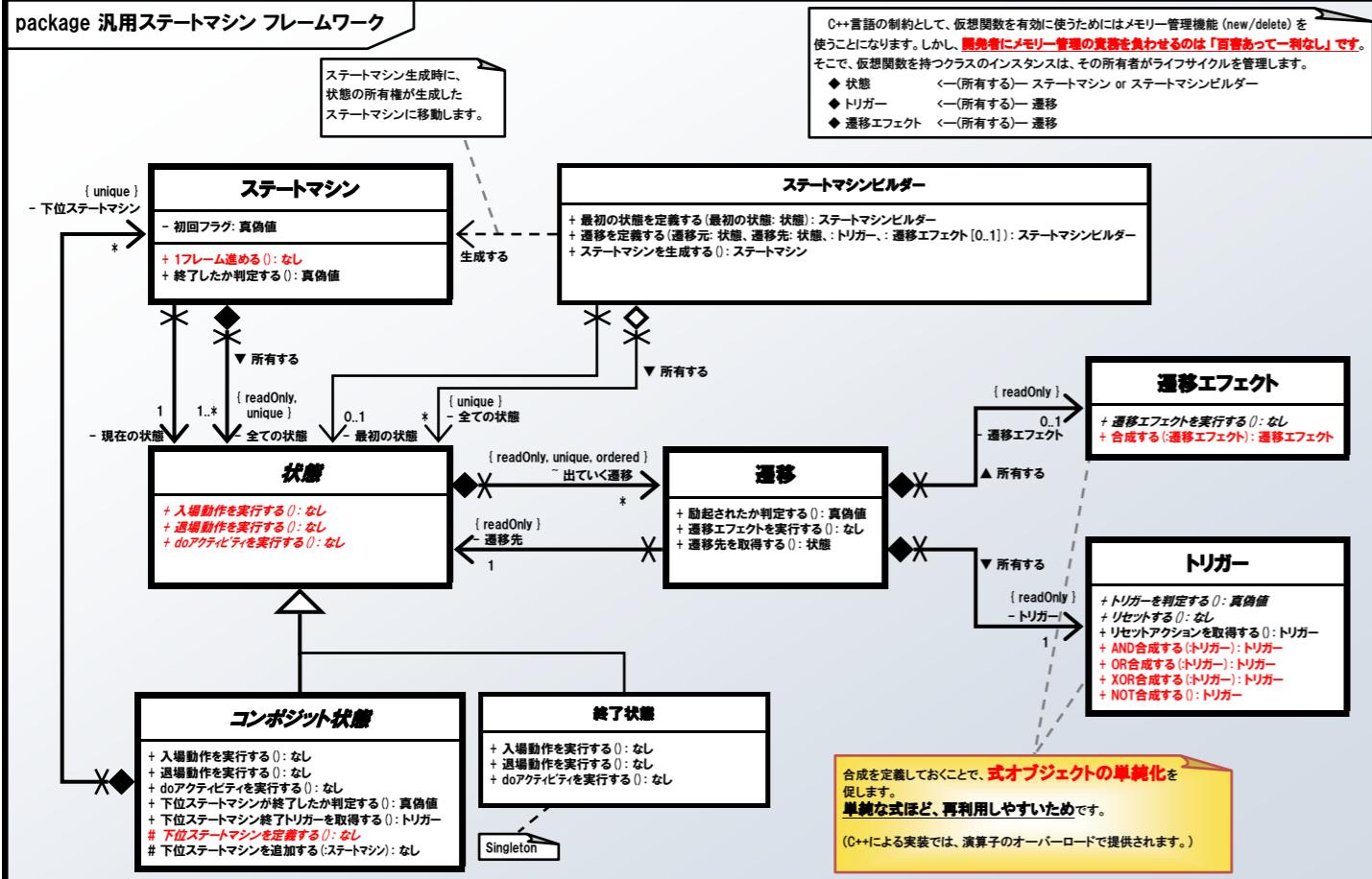
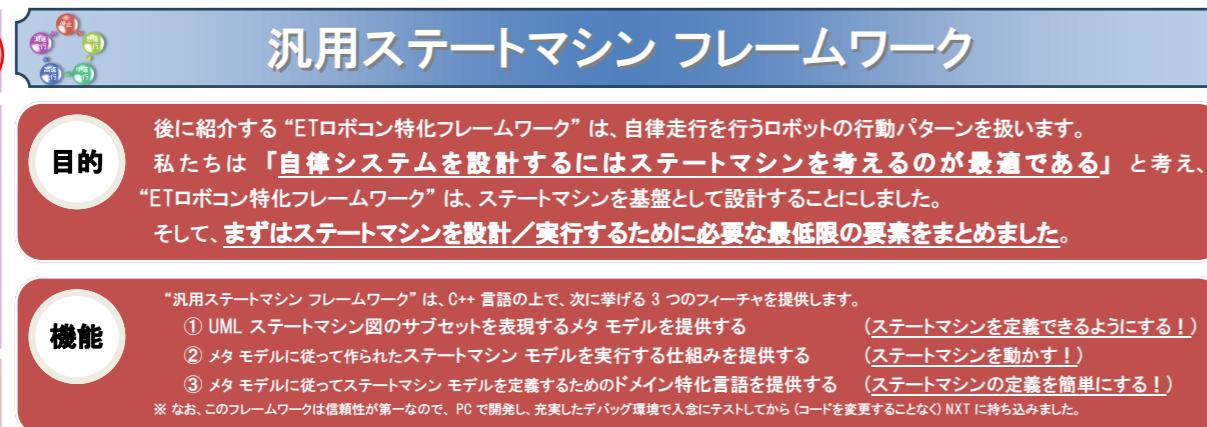
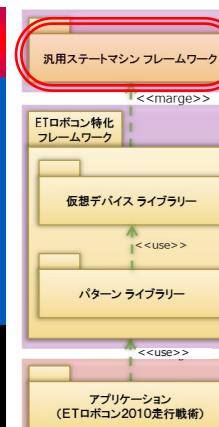
2009年度のフレームワークを見直し、C++言語で再実装します。



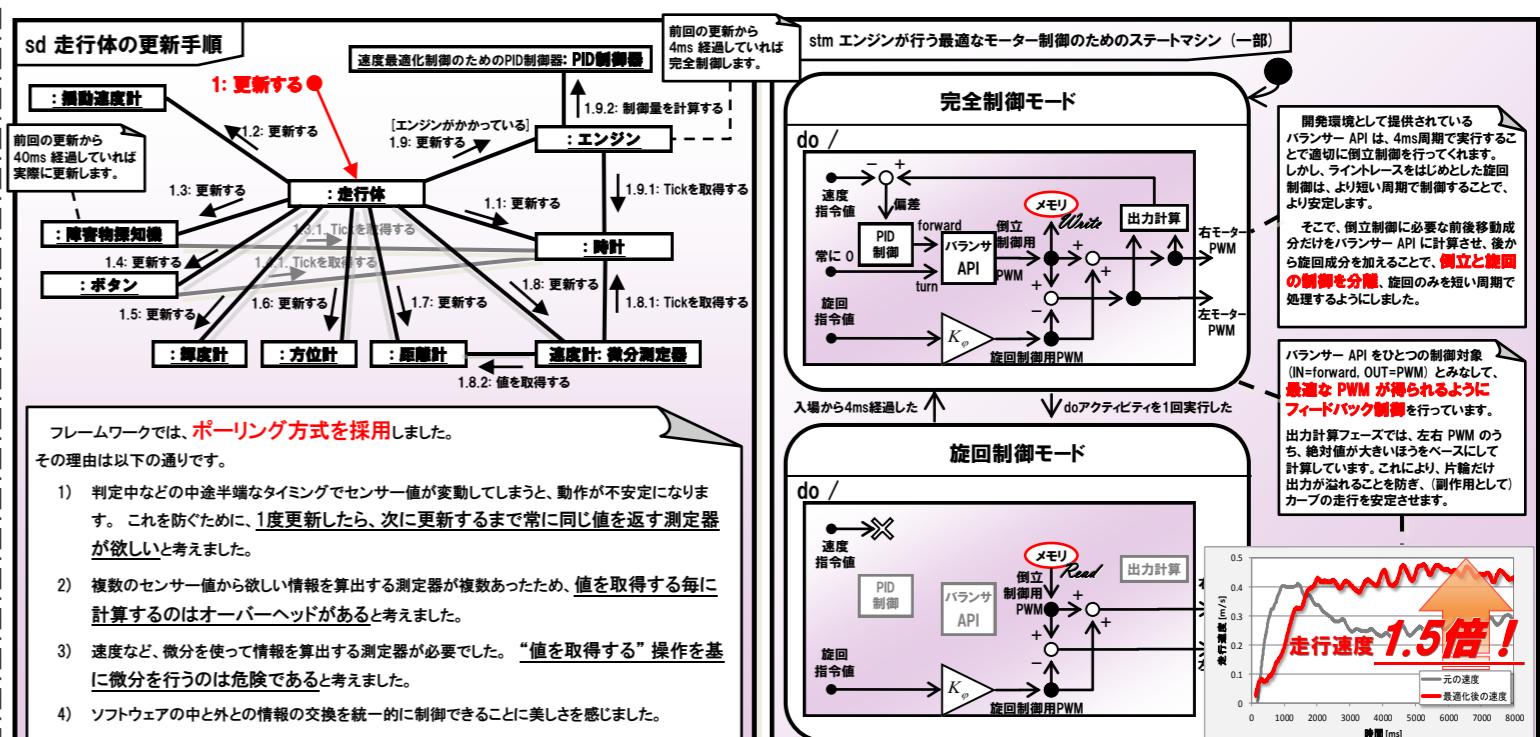
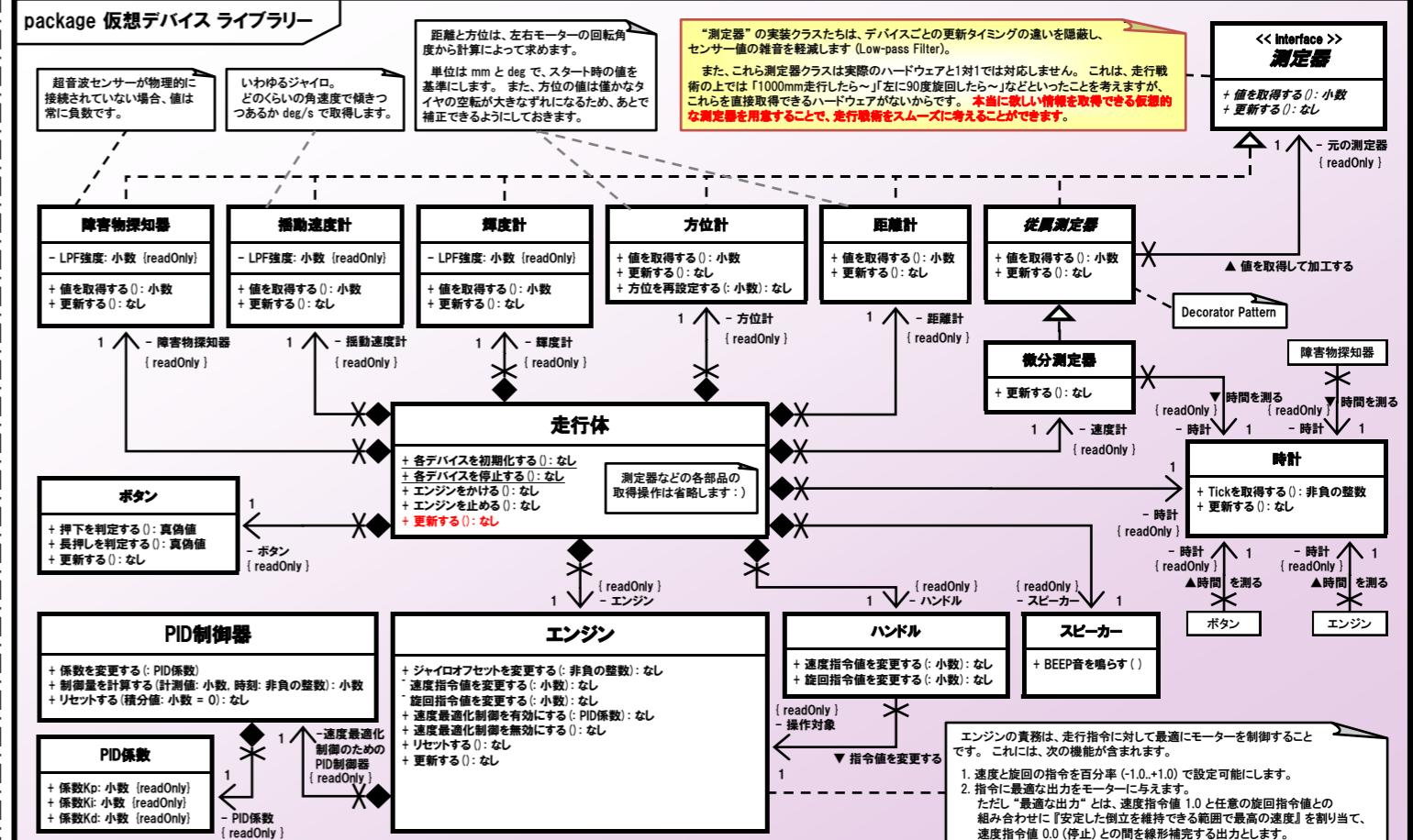
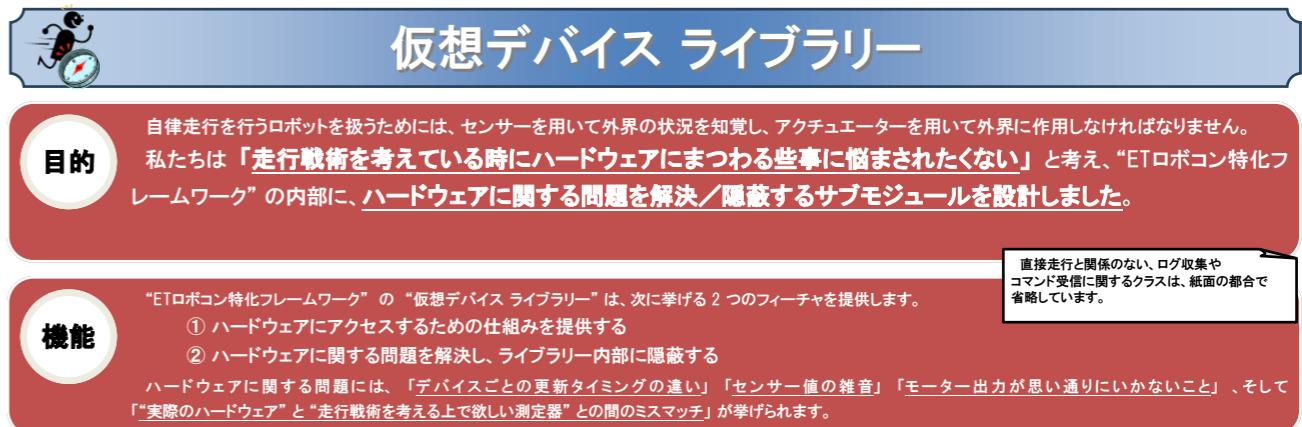
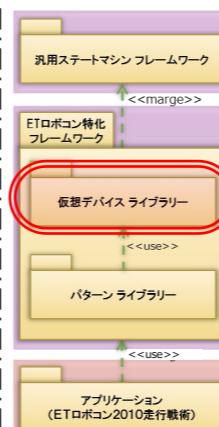
uc 走行体と監視制御システムのユースケース

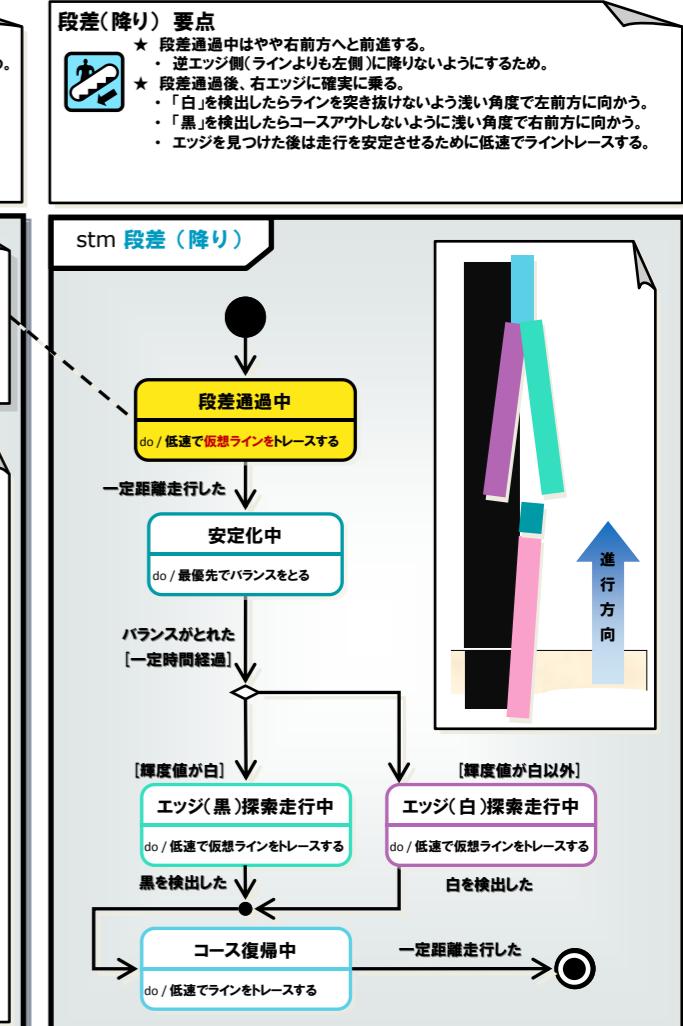
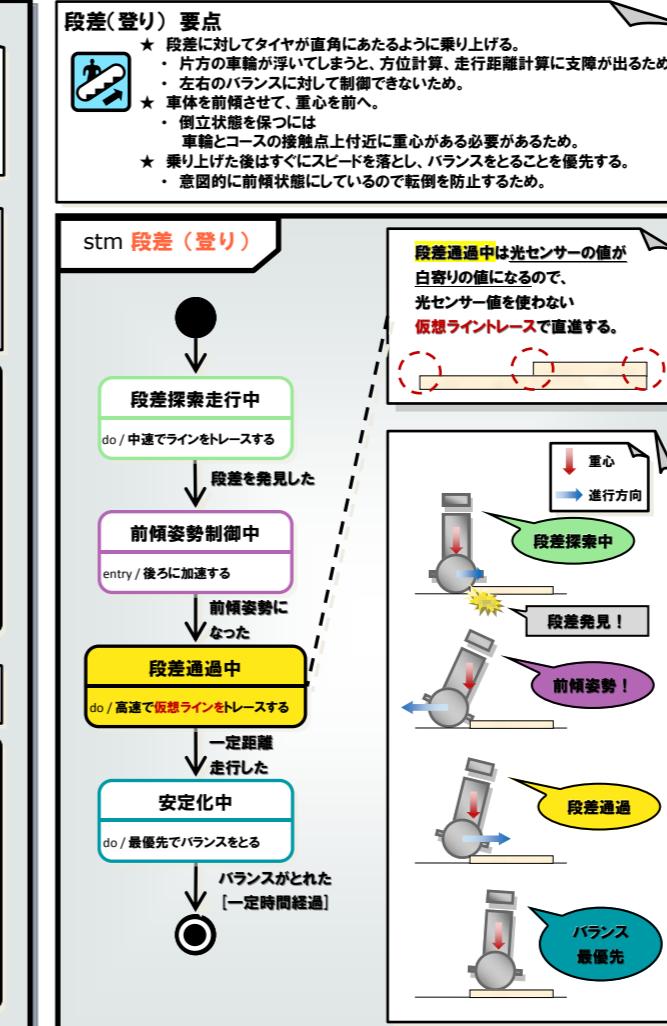
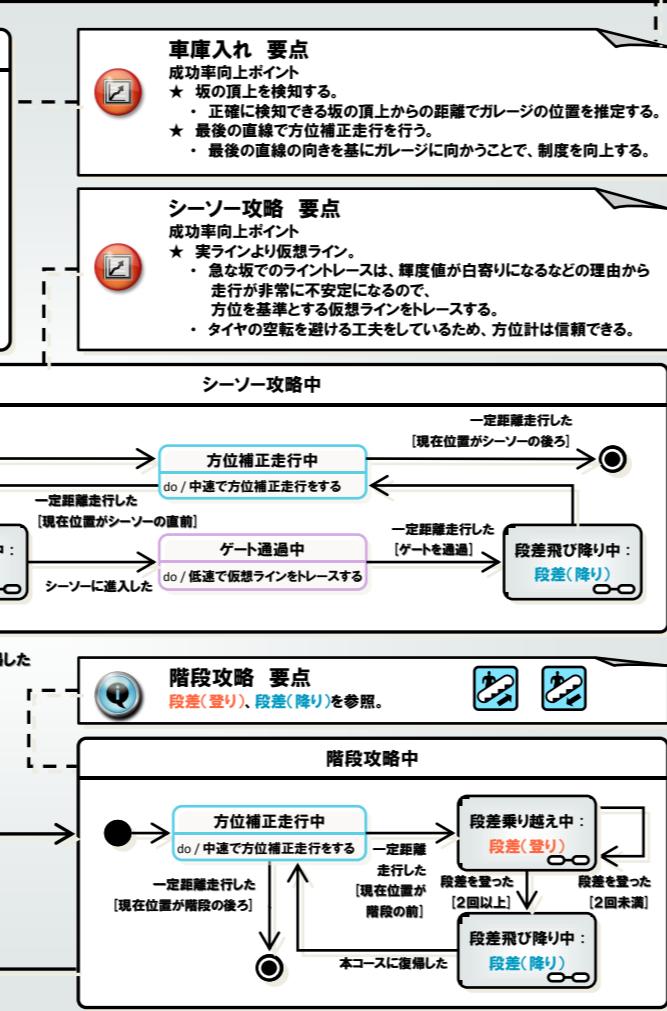
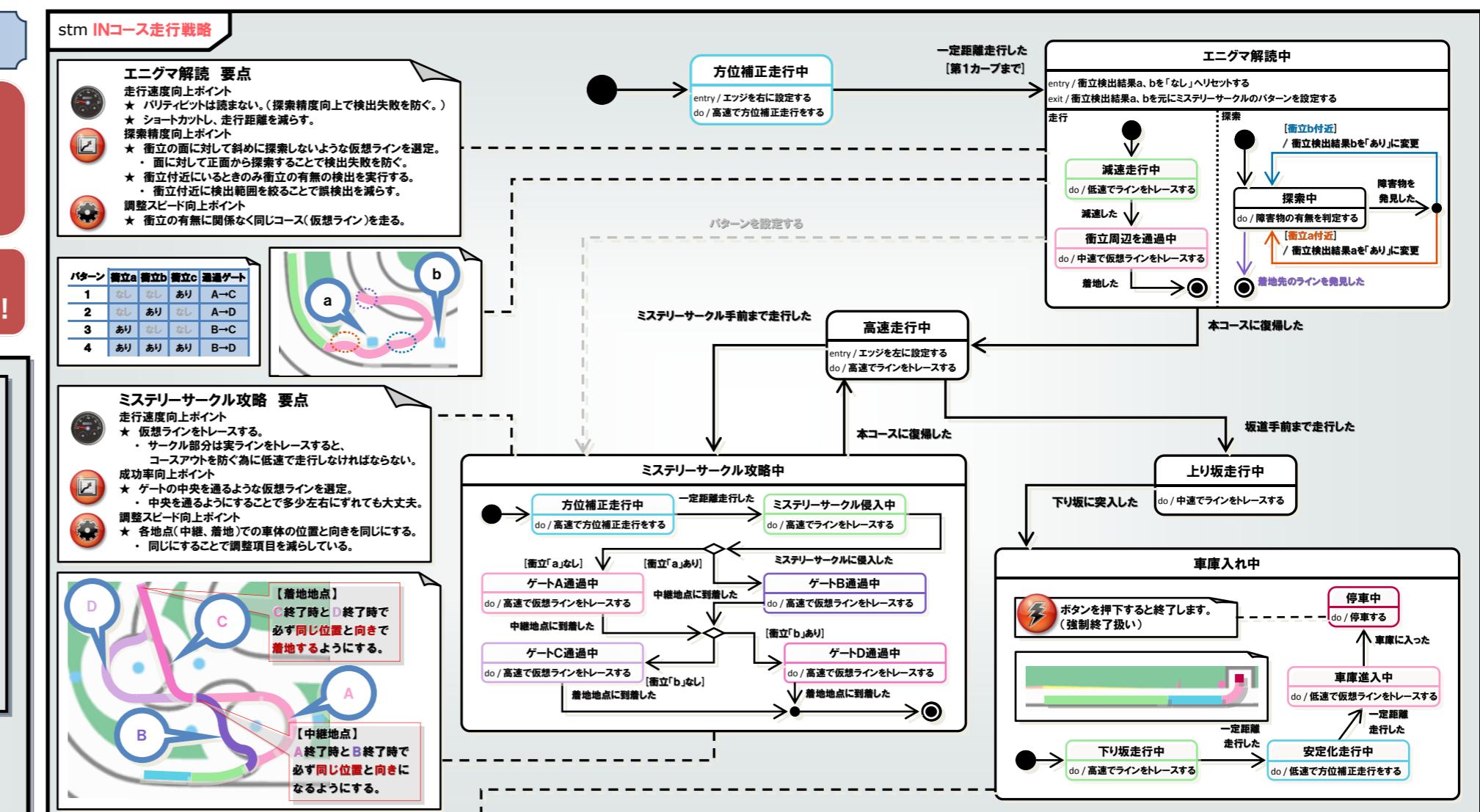
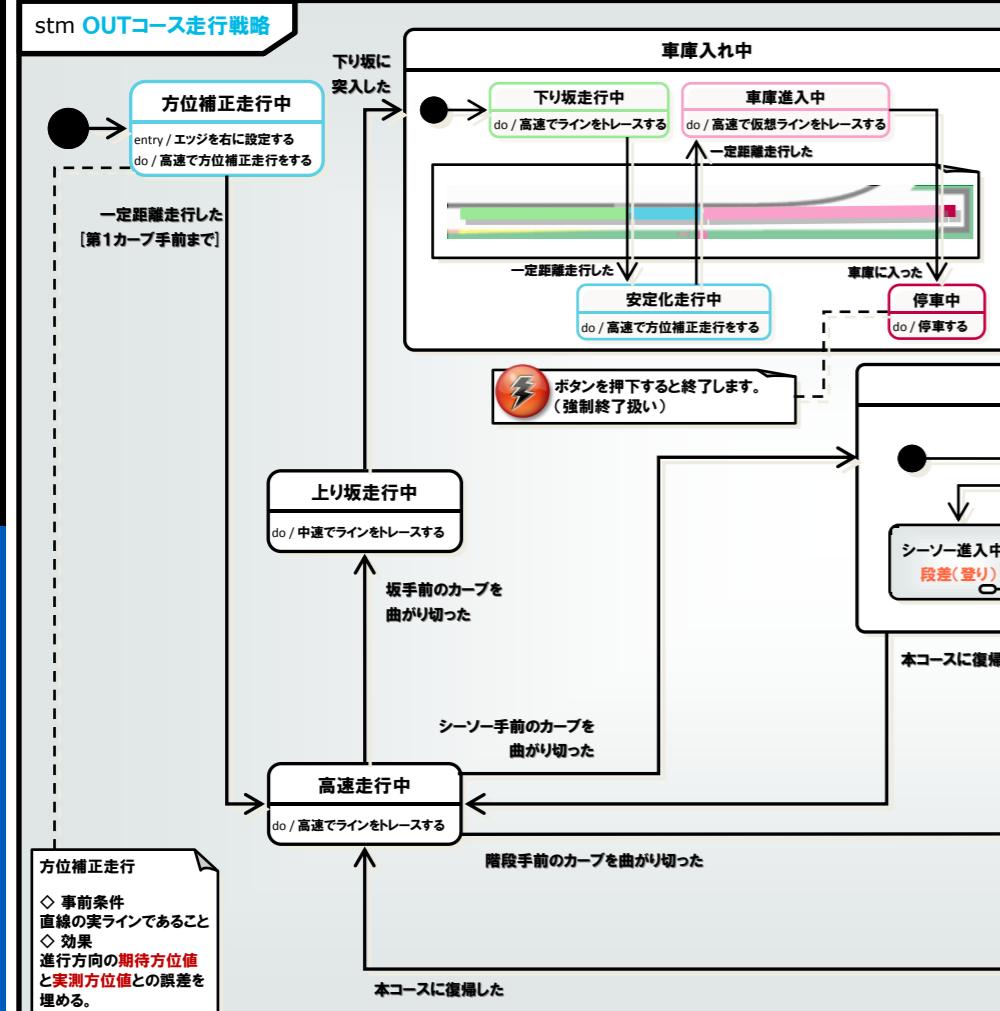
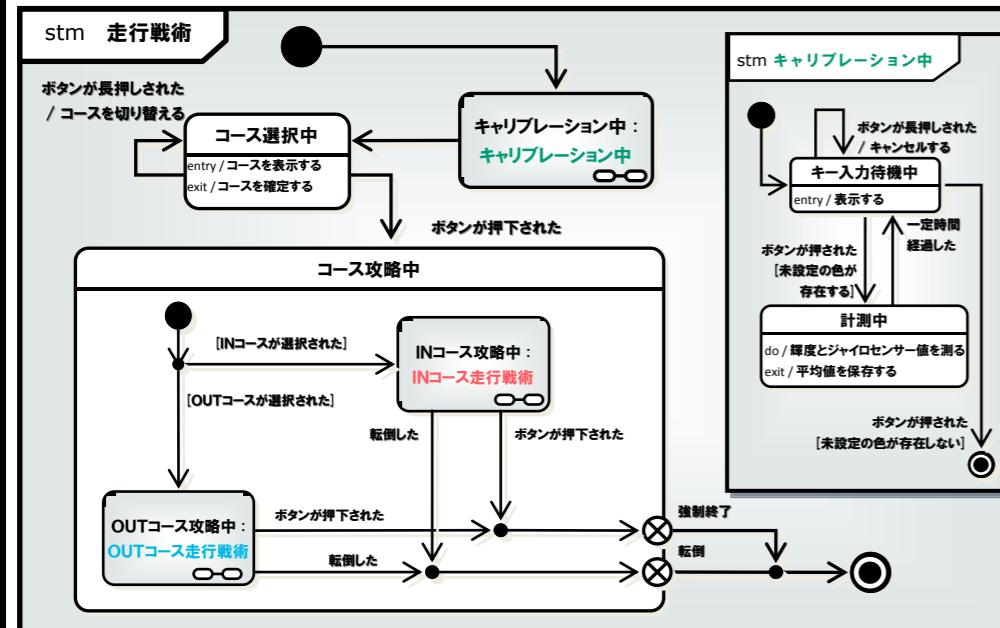


汎用ステートマシン フレームワーク



仮想デバイス ライブラリー



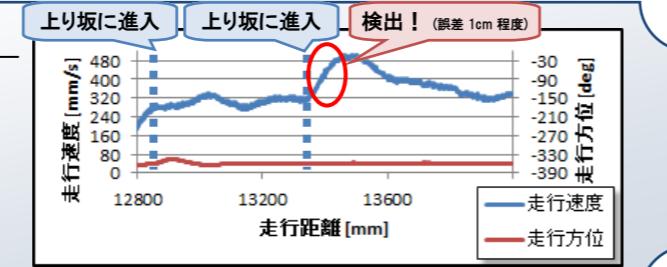


坂の頂上を検出する

ガレージインを確実に成功させるためには、2つの技術が必要です。

- ① 目的の軌道を正確に走行すること。
⇒ 仮想ラインを用いることでクリアしました。
- ② ↑ の基準となる位置を正確に見つけること。
⇒ **坂の頂上を検出することでクリア**しました。

頂上を越えた直後に発生する“速度”的な波形を検出します。

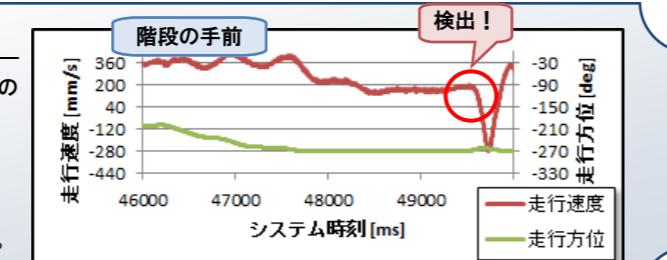


段差を検出する

今年度最大の難所は、“段差を登る”ことです。そして、段差を登るための制御を正確に行うためには、段差があることを知らなければなりません。

⇒ **速度を監視することで、段差に触れたことを検出します。**

段差に接触した直後に発生する“速度”的な波形を検出します。

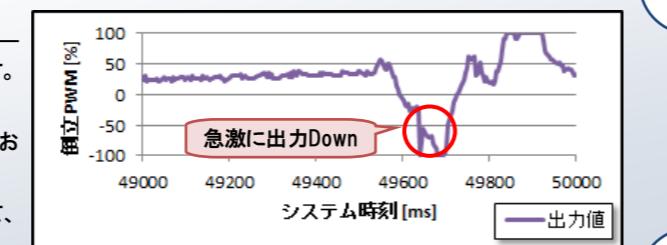


ジャイロ オフセット操作による急加減速

車体を前傾／後傾にするためには、加速度を与える必要があります。特に、段差を登るために、段差を検出後、即座に車体の傾斜を制御します。

しかし、速度指令値には、バランサー API 内で強力な LPF が掛けられており、指令値を変えてから加減速するまで時間がかかります。

そこで、バランサー API の制御ロジックを逆手にとって、**ジャイロ オフセットを操作することで、即座に加速度を与えます。**

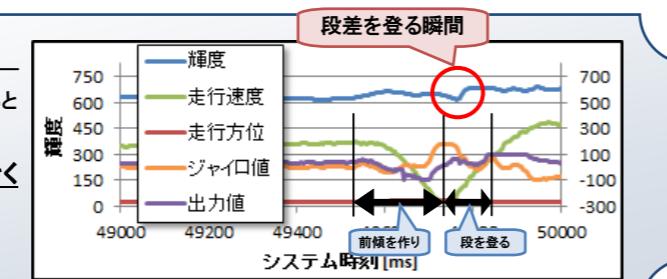


段差を登る／降りる瞬間に仮想ラインを使う

光センサーが車輪よりも前方についているため、段差を登るとき／降りるときには、光センサーと地面との距離が変化します。

これが原因で、**段差の登り降りを行う瞬間、光センサーの値が大きく変動してしまいます。**

その対策として、**光に頼らない仮想ラインを利用して走行します。**



追加課題: 並行性設計について

私たちは、並行性について2種類に大別できると考えました。

- 1) ハードウェア特性に対応するための並行性
- 2) 走行戦術を考える上で必要な並行性

そして、これら並行性を、次のようなコンセプトで設計に反映してきました。

- 1) 「ハードウェア特性に対応するための並行性」は、「仮想デバイスライブラリ」が隠蔽する
⇒ See also right in p.2
- 2) 「走行戦術を考える上で必要な並行性」を実現する仕組みを「ステートマシンフレームワーク」が提供する
⇒ See also left in p.2
- 3) 可能な限りタスクを分割しない
⇒ 結果として、アプリケーションではタスクを1つしか定義していません

◆「可能な限りタスクを分割しない」とは?

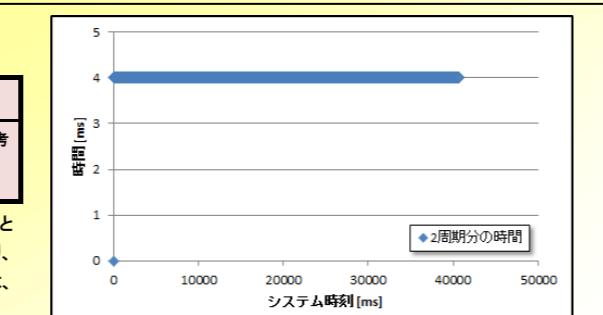
NXTはシングルコアCPUが動かしているため、タスク分割によるメリットはかなり少ないです。

メリット	デメリット
●複数の直交する処理が存在し、かつそれらが必要な時間内に完了しない場合、自動的に優先度の高いタスクが実行されるようにスケジュールされる。	●タスク間同期や共有データアクセスに関する問題を考える必要が追加される。 ●タスク切り替えのオーバヘッドが追加される。

各センサーやモーターの制御を個々に見るとそれらの処理は直交していますが、全体を見ると「センサー値の取得 ⇒ 状態遷移の処理 = モーターへの出力更新」という流れがあります。つまり、どこかをタスク分割しても、全体の流れを変えないためにタスク間同期が必要になります。それでは、逐次実行と変わりありません（もちろん、マルチコアなら直交なロジックを分割統治することで性能向上のメリットがあります）。

また、メインループの1サイクルの処理は、じゅうぶんに時間内に収まっています。

使うメリットがなく、被るデメリットが非常に厳しいため、私たちはタスク分割を行いません。



↑収集したログの各レコード(2周に1回送信)の tick の差を計算してグラフにしました。レコード間の時差がバラけてないので、すべての処理が時間内に収まっています。

ET ロボコン走行体 監視制御システム

目的

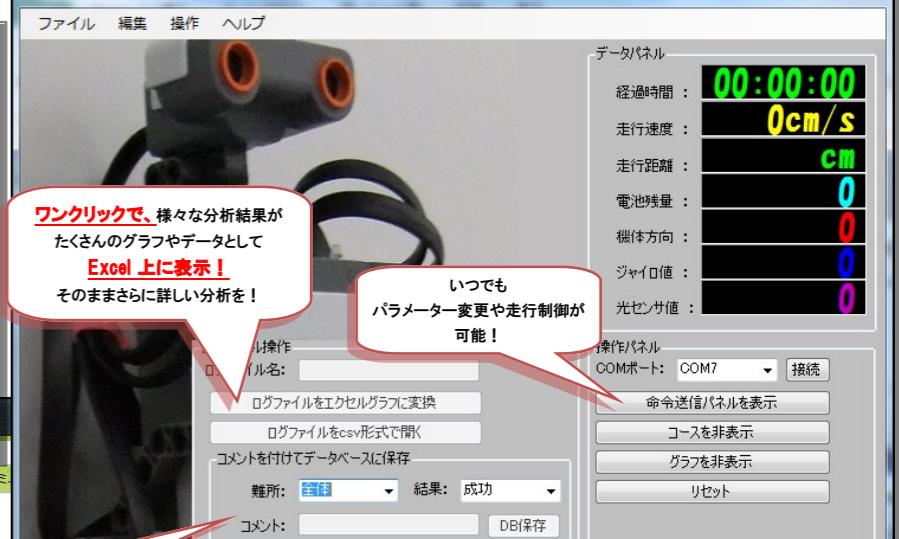
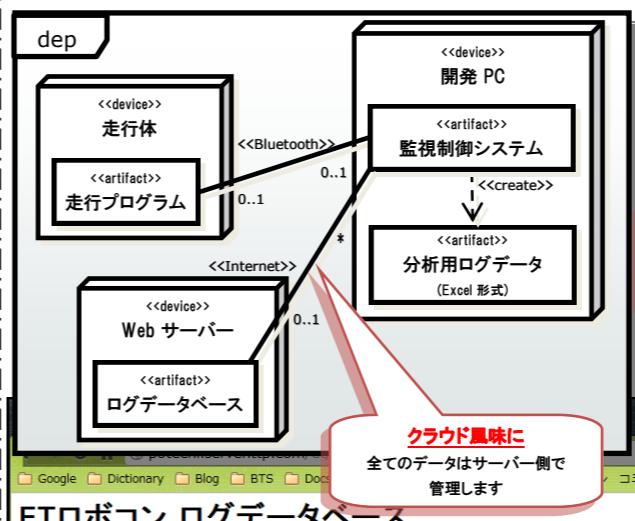
デバッグや走行戦術の妥当性検証に必要なもの、それは情報です。／また、走行体へのプログラムアップロードは手間のかかる作業です。私たちは「これで良い」と確信に足る、あるいはダメだと確認できるだけの情報を収集する仕組みが必要だ」「数多い制御パラメーターを調整する毎に再コンパイルしたくない」「試走中に迷走した車体をすぐに回収したい」と考え、これらを実現する監視制御システムを構築しました。

機能

ETロボコン走行体 監視制御システムは、次に挙げる7つのフィーチャを提供します。

- ① 走行中の走行体から、内部状態に関するログデータを受信／記録する
- ② 記録しているログデータをリアルタイムにグラフやアニメーションで表示する
- ③ 記録されているログをさまざまな計算やグラフを用いて解析する

- ④ ログデータと解析結果を注釈つきで履歴に残す
- ⑤ 履歴に残された情報を多数のPCで共有する
- ⑥ 制御パラメーターを走行体に送信／変更する
- ⑦ 走行体の走行をコントローラーで制御する



アップロード日時	ログ取得PC	離所	結果	コメント	ファイル名	ダウンロード	動画	削除
2010/09/16 19時23分	F32	【階段】	【成功】	とりあえず越えました。状態遷移が段内で終わってた	20101013_205507.csv	DOWNLOAD	-	削除
2010/09/16 19時18分	F32	【階段】	【成功】	ジャイロオフセット変更無し	20101013_200822.csv	DOWNLOAD	-	削除
2010/09/16 19時17分	F32	【階段】	【成功】	距離が変わった気がしない。あと↓成功	20100916_192853.csv	DOWNLOAD	-	削除
2010/09/16 19時02分	F32	【階段】	【失敗】	2段目で倒れました。(降りるときの距離減)電池BB	20100916_192533.csv	DOWNLOAD	-	削除
		【階段】	【失敗】	スタートをミスした。右の緑のほうに着地	20100916_192147.csv	DOWNLOAD	-	削除
		【階段】	【成功】	電池8号でジャイロ500Hz	20100916_191751.csv	DOWNLOAD	-	削除
		【階段】	【成功】	電池の残量によってジャイロを変える必要あり(? 90)で500	20100916_191440.csv	DOWNLOAD	-	削除
		【階段】	【失敗】	上れない！	20100916_190208.csv	DOWNLOAD	-	削除

注釈とともに皆で共有！振り返っての確認にも役立った！



Column: C++言語による開発について

私たちは今回、C++言語を用いて開発を行いました。これが地区大会の審査の中で物議を醸したうなので、少し掘り下げてお話ししたいと思います。

Q1.なぜC++言語で開発しようと思ったの？

昨年度まではC言語で実装していましたが、やはりオブジェクト指向で設計したモデルをC言語で実装するというは無理があって、いろんなところが歪んでいました。特に昨年度は継承（あるいはジェネリクス）に相当することを実現しようと、関数ポインタが乱舞し、void* データが飛び交う素敵なコードになっていました（汗）。これではいけないと、今年はC++言語で実装することにしました。

Q2. C++言語で開発してどうだった？

開発はずいぶん楽になりました。先ほど言ったような酷いコードは、即ちバグの温床です。それが、C++の型システムを有効に使うことによって、コンパイラが殆どすべてのバグを弾いてくれるようになりました。おかげで、開発中にバグに悩むことは一切無かったです。

Q3.メモリーが足りなくなったりしなかった？

“あっ”という間に足りなくなりました（笑）。1週間もたなかつです。ステートマシンフレームワークの実装だけで拡張ファームウェアには乗らなくなってしまって、NXT BIOSを使って開発を続けました。でも、NXT BIOSの上では、まだ余裕がありますよ。半分も使ってません。

Q4.具体的にはいくつぐらい？

実行イメージがIN/OUT合わせて92KBです。このうち、76KBくらいがフレームワークです。RAMは調べていませんが、使用するRAMが可能な限り少なくて済むように、フレームワーク内で工夫しています（2ページ左側）。

それについて、懇親会で「こんなところでnew/deleteして大丈夫なのか」と質問されました。その場では「もちろん大丈夫です」と答えた気がしますが、本当は「大丈夫でした」と答えるべきでした。私には「大丈夫じゃないかもしれない」という意識がなかったので。

Q5. C++言語で開発して一番良かったことは？

良かったことはたくさんあります。1番は、その強力な型システムを使えたことでしょう。メンバ関数や演算子のオーバーロードのおかげで直感的で分かりやすいフレームワークでできましたし、デストラクタと情報隠蔽を使ってリソース管理をフレームワークに閉じ込めるこどもできました。「ポインタが危険なら安全なポインタを作れば（使えば）いい。配列が危険なら安全な配列を作れば（使えば）いい。」という普通のC++erの思想を体现できたと思います。これはC言語ではできなかったことです。

Q6. C++言語で開発して一番大変だったことは？

今回は実行イメージのサイズが心配で、テンプレート機構を封印していました。C++言語だから大変だったという訳じゃないですが、テンプレートが使えないかったのは大変でした。例えば、コンテナなどを自分で作ることになったわけです。今にして思えば、最初からテンプレートをちゃんと使って、問題が出たら対策を考えればよかったですね。