組込みソフトウェアの開発支援に関する研究
―制御パターンを利用したプログラムの自動生成の考察―

2011SE279 上田哲彰　2011SE287 XU Xinping
指導教員：沢田篤史

1　はじめに

近年, 我々の身の回りには, 携帯電話, 自動車などを制御する組込みソフトウェアが数多く存在している. 組込みソフトウェアの開発では, デバイスごとにソフトウェアを開発する必要がある. 組込みソフトウェアの大規模化, 複雑化が進み, 開発のコストが増大している. 開発のコストの増大に伴い, 開発の効率化が求められている. ソフトウェアの開発の工程は基本設計, 詳細設計, 製作. 結合テスト, 総合テストに分けられ, 各工程の中で製作が最も比率が大きい [3].

多くの組込みソフトウェアは単純な制御を行なっているがデバイスの種類ごとに一からプログラムの開発をしている場合もある. 一般的に, 開発者はデバイスの種類ごとに設計を一から作るのではなく, アーキテクチャパターンやデザインパターンを利用することで開発の効率化を行う.

本研究の目的は, 組込みソフトウェアにおけるソフトウェアの自動生成による開発支援である. 組込みソフトウェアの構造を調査し, 組込みソフトウェアにおける典型的な制御パターンを定義する. その典型的な制御パターンに基づいたソフトウェアについて調査, 検討し, そのパターンに基づいてプログラムを自動生成することで, 開発の効率化を行なう.

本研究では, 組込みソフトウェアにおける典型的な制御パターン定義し, 典型的な制御パターンの分析に基づいて制御仕様を記述する方式を提案する. さらに, この記述方式に基づいて記述された仕様から組込みソフトウェアのソースコードを自動生成するために, アーキテクチャの定義と自動生成ツールの開発を行う. 事例として LEGO Mindstorms EV3 を制御するソフトウェアを扱う. アーキテクチャ定義では, 組込みデバイスに多いセンサとアクチュエータを制御するパターンに基づき, いくつかのデザインパターンを組み合わせて定義する. 自動生成ツールはモデル駆動型アーキテクチャに基づく自動生成ツールを作成した. 出力は Java のプログラムコードで, 入力は仕様表, 決定表である.

2　背景技術

2.1　センサ・アクチュエータパターン

センサ・アクチュエータパターンとは組込みソフトウェアでよく使われる, 制御構造にかかわるアーキテクチャパターンである. 外界のイベントをセンサでとらえ, それに基づいた処理をし, アクチュエータを介して外界に応答するパターンである. 組込みシステムで多いリアクティブなシステムの典型的な構造である [4]. 図 1 にセンサ、アクチュエータパターンにしたがったアーキテクチャの例を示す. センサデバイスからの情報をセンサ制御が集め, データ処理は集めた情報を処理してどう応答するかを判断し, アクチュエータ制御がその判断に基づいてアクチュエータデバイスを駆動する.

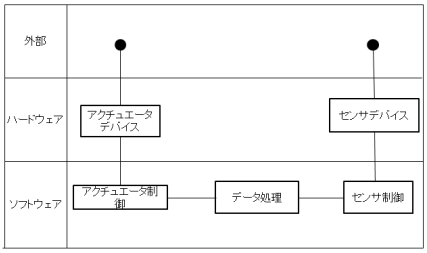

図 1　センサ・アクチュエータパターン

2.2　モデル駆動型アーキテクチャ (MDA)

モデル駆動型アーキテクチャ [1]（以下 MDA）はモデルを中心としたソフトウェア開発の手法である.MDA はプラットフォームに依存しないソフトウェア開発を目的とする.MDA ではプラットフォームに独立モデル (PIM) の設計を行い. プラットフォームに依存したモデル (PSM) への変換を行う. 複数のプラットフォームに対応したプログラムコードの自動生成が可能となる.

3　典型的な制御パターンの分析

組込みシステムは一般に, センサ, アクチュエータを持つ. センサとアクチュエータを制御する典型的な制御パターンとして, センサの値でアクチュエータの動作が一意に決まるソフトウェアが挙げられる. 本研究では, この典型的な制御パターンで制御される一般的なセンサ, アクチュエータの使い方について整理, 分類した. 典型的な制御パターンに基づいたアーキテクチャパターンとしてセンサ・アクチュエータパターンを利用する.

3.1　センサとアクチュエータの使用パターン

典型的な制御パターンにおいて一般的なセンサ, アクチュエータを制御する際, 制御モジュール (クライアント) とのやりとりのパターンをセンサ, アクチュエータそれぞれについて分類し, 使用パターンとして定義する.

センサとクライアントとの通信パターンに基づいて以下の 4 つに分類した.

- 同期条件なしパターン

  クライアントはセンサに対し, 取得する命令を送る. センサはその命令を受けて外部情報を取得し, 取得した

外部情報をクライアントへ返す. 図 2 は同期条件なしパターンのシーケンス図である.

- 非同期条件なしパターン
  クライアントはセンサに対し, 取得する命令を送る. センサはその命令を受けて外部情報を取得し, クライアントが取得停止命令を出すまで通知を行なう. 図 3 は同期条件なしパターンのシーケンス図である.
- 同期条件ありパターン
  クライアントはセンサに対し, 条件の登録をする命令を送る. センサはその条件にあった外部情報のみをクライアントへ返す. 図 4 は同期条件なしパターンのシーケンス図である.
- 非同期条件ありパターン
  クライアントはセンサに対し, 条件の登録をする命令を送る. センサはその条件にあった外部情報を取得した際, クライアントへ通知を行なう. 図 5 は同期条件なしパターンのシーケンス図である.

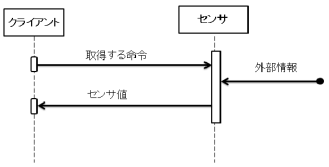

図 2 同期条件なしパターンのシーケンス図

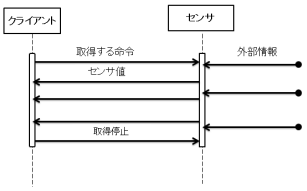

図 3 非同期条件なしパターンのシーケンス図

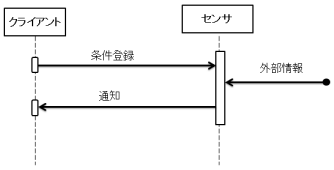

図 4 同期条件ありパターンのシーケンス図

同様にアクチュエータの使用パターンを以下の 2 つに分類した.

- 同期パターン
  クライアントからアクチュエータに対し, 動作する命令を送る. アクチュエータはその命令を受けて外部出力し, 動作の完了をクライアントへの通知を行なう. 図 6 は同期パターンのシーケンス図である.

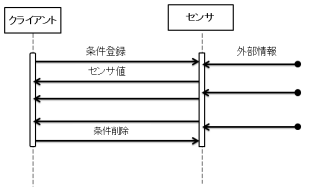

図 5 非同期条件ありパターンのシーケンス図

- 非同期パターン
  クライアントからアクチュエータに対し, 動作する命令を送る. アクチュエータはその命令を受けて, クライアントから動作の停止命令があるまで外部出力し, 停止命令を受けとった場合, 動作の完了をクライアントへの通知を行なう. 図 7 は非同期パターンのシーケンス図である

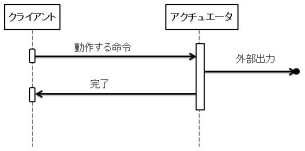

図 6 同期パターンのシーケンス図

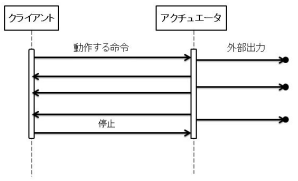

図 7 非同期パターンのシーケンス図

3.2 Observer パターンの適用

センサ・アクチュエータパターンのセンサ制御とデータ処理に適用する. 使用パターンのセンサの非同期条件なしパターン, 非同期条件ありパターンの時に使う.Observer パターン [2] を適用することにより, 状態変更の通知をセンサ制御から送ることができるようになり, センサとデータ処理の構造を分離できる.

3.3 Command パターンの適用

センサ・アクチュエータパターンのアクチュエータ制御とデータ処理に適用する.Command パターン [2] を適用することでセンサが取得した情報をまとめてアクチュエータ制御で扱える.

4 自動生成の仕様記述とツールの実装

MDA に基づく自動生成をするために 3 章で定義したパターンに基づいて自動生成の入力の仕様記述を定義した. センサ仕様表, アクチュエータ仕様表, 決定表を自動生成

の入力とする. プラットフォームコードとしてアーキテクチャ, ひな型,EV3 のクラスライブラリ, 対応表を定義する. 図 8 はアプリケーションアーキテクチャである. 図 9 は自動生成されるプログラムのクラス図である.

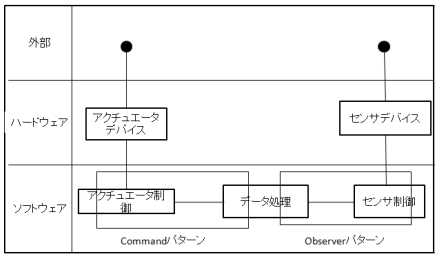

図 8 アプリケーションアーキテクチャ

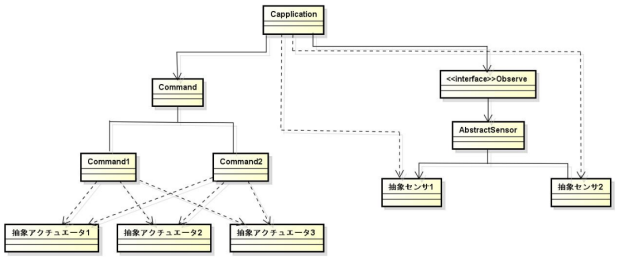

図 9 自動生成されるプログラムのクラス図

4.1 可変部分と不変部分の分類

自動生成されるプログラムコードを不変部分と可変部分に分類する. 不変部分は自動生成されるプログラムコードに共通しているのでひな型として作る. 可変部分は入力を元に生成する. 可変部分は図 9 の抽象センサ,Command, 抽象アクチュエータ,CApplication が持つ.

4.2 センサ仕様表の定義

表 1 はセンサ仕様表である. センサ仕様表は抽象センサクラスを生成するのに必要な情報の入力するために定義する. 抽象センサ名, 利用パターン, 物理センサクラス, データ取得メソッド, センサ値型, 優先度を持つ. 抽象センサ名は抽象センサクラスの名前になる. 利用パターンはセンサの使用パターンで定義された 4 つのパターンから選択される. 物理センサクラスはハードウェアセンサのもつクラスである. データ取得型メソッドは LEGO Mindstorms EV3 のクラスライブラリから選択する. センサ値型は物理センサクラスの返り値をデータ処理に渡す時の型で, 優先度はセンサを処理する順番である.

表 1 センサ仕様表

抽象センサ名	利用パターン	物理センサクラス	データ取得メソッド	センサ値型	優先度
TS	同期条件なし	EV3TouchSensor	getValue	boolean	1
CS	非同期条件なし	EV3ColorSensor	getValues	int	3
USS	非同期条件あり	EV3UltraSonicSensor	getValues	float	2

4.3 アクチュエータ仕様表の定義

表 2 はアクチュエータ仕様表である. アクチュエータ仕様表は抽象アクチュエータを生成するのに必要な情報の入力するために定義する. 抽象アクチュエータ名, 利用パターン, 物理アクチュエータ, 動作, 優先度を持つ. 抽象アクチュエータ名は抽象アクチュエータクラスの名前になる. 利用パターンはアクチュエータの使用パターンで定義された 2 つのパターンから選択される. 物理アクチュエータクラスはハードウェアアクチュエータのもつクラスである. 優先度はアクチュエータを動作させる順番である. 動作は自然言語で記述し,LEGO Mindstorms EV3 の物理センサがもつ動作に結びつける. 一般的なアクチュエータの外部へ動作として動く, 光る, 鳴らすの 3 つに分類した. 分類したものをアクチュエータが持つ動作メソッドと結びつけ入力の一部として適用する. アクチュエータの動作を自然言語で入力することで, モデルを自動生成する環境から離すことができる. 表 3 は自然言語で書かれた動作とアクチュエータの動作メソッドとの対応表である.

表 2 アクチュエータ仕様表

抽象アクチュエータ名	利用パターン	物理アクチュエータクラス	動作	優先度
MT	同期	RegulatedMotor	動く	1
MA	同期	BaseMotor	動く	2
LCD	非同期	LCD	光る	3

表 3 自然言語に対する動作メソッドの対応表

自然言語の動作	EV3 のメソッド
動く	メソッド 1
光る	メソッド 2
鳴らす	メソッド 3

4.4 決定表の定義

表 4 は決定表である. 決定表はセンサの値に対するアクチュエータの動作を決定する入力のために定義する. 入力か出力か, 抽象クラス名, 状態, パターンを持つ. 入出力はセンサなら入力, アクチュエータなら出力である. 抽象クラス名にそれぞれのクラス名を入力する. センサは状態にセンサの値の条件を入力し, 各パターンはその状態が Y か N かを入力する. アクチュエータのパターンはどのように動作するか自然言語で入力する. 入力は自然言語に対するメソッドの対応表に基づいてアクチュエータの動作メソッドと引数に変換する.

4.5 MDA ツールの設計

典型的な制御パターンに基づくソフトウェアを自動生成する MDA ツールの設計を行なった. 入力とプラットフォームコードとして図 10 に示したアーキテクチャ, ひな型,EV3 Mindstorms EV3 のクラスライブラリをあわせて Java のプログラムコードを出力する. 図 10MDA ツールのアーキテクチャである.

表 4 決定表

入出力	抽象クラス名	状態	パターン 1	2	3	4	5
入力	TS	on	N	Y	Y	Y	Y
入力	TS	off	Y	Y	N	Y	N
入力	CS	黒	Y	Y	N	Y	N
入力	CS	赤	N	N	Y	N	N
入力	CS	緑	N	N	N	N	Y
入力	CS	その他	N	N	N	N	N
入力	USS	≦ 5	N	N	N	Y	N
入力	USS	5 <	Y	Y	Y	N	Y
出力	MT		0	40	0	40	100
出力	MA			逆に 100			
出力	LCD		表示				

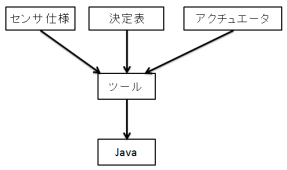

図 10 MDA ツールのアーキテクチャ

4.6 ひな型

本研究で作成した自動生成ツールのひな型の例を以下に記述する.

```
public class 「抽象センサ名」 extends AbstractSensor{
 private 「物理センサ名」 S;
 private 「センサ値型」[] buf = new 「センサ値型」[2];
 「抽象センサ名」(){
  S = new 「物理センサ名」(SensorPort.S1);
  buf[1] = -1;
 public 「センサ値型」 getValue(「センサ値型」 i){
  buf[0] =S.「データ取得メソッド」();
  return buf[0];
 }
 public void run() {
  buf[0] = getValue(buf[1]);
  if( buf[1] != buf[0]) {
   notifyS();
   buf[1] = buf[0];
   }
  }
 }
}
```

図 11 抽象センサクラスのひな型

5 考察

5.1 典型的な制御パターンに基づいたソフトウェアのアーキテクチャ設計に関する考察

本研究で定義した典型的な制御パターンとそれに基づくソフトウェアのアーキテクチャ設計による開発の効率化を考察する. 本研究では, 典型的な制御パターンに基づいたソフトウェアにアーキテクチャパターンとセンサ, アクチュエータを制御するのに適したデザインパターンを適用した. 典型的な制御パターンを定義し, それに基づいて設計したことで, 典型的な制御パターンに基づくプログラムの設計を一からする必要がなくなり, 設計にかかる時間を削減できた. 加えて, 一般的なセンサ, アクチュエータについて整理することでセンサとアクチュエータの使い方を定義することで様々なセンサ, アクチュエータを持つデバイスに対応することができた.

5.2 仕様記述の妥当性に関する考察

MDA に基づく自動生成ツールを作成し, その入力として定義した仕様記述について妥当性を考察する. 自動生成されるプログラムコードの分析を行ない可変部分, 不変部分に分類した. 可変部分を生成する入力の仕様記述を定義する必要があると考えた. 入力としてセンサ仕様表, アクチュエータ仕様表, 決定表を定義した. 仕様記述を定義したことにより自動生成するのに必要な情報を整理することができたと考える. アクチュエータの動作を自然言語で記述し, 自動生成ツールで自然言語とアクチュエータの動作メソッドを結びつけることで, 入力をよりモデルに近づけることができたと考える.

6 おわりに

本研究では, 組込みソフトウェアにおける, 典型的な制御パターンに基づくソフトウェアの自動生成による開発支援を提案した. 典型的な制御パターンに基づいたアーキテクチャパターン, デザインパターンを適用したソフトウェアとそのソフトウェアの自動生成に必要な仕様記述方式を定義した. 定義に基づく自動生成することができた. 自動生成ツールの作成し, 事例検証を行なった. 自動生成による開発支援で開発の時間を削減することができたと考える.

今後の課題として本研究で定義した使用パターン, アーキテクチャ, 仕様記述はごく簡単なソフトウェアを典型的な制御パターンと定義し, それに基づいて定義されている. 例えば使用パターンでは取得するのに時間の制約があるときなど複雑な制御をするものにも対応する必要がある. センサの使用パターンではセンサの値を通知する条件がある場合を定義したが, 条件が満たされない限り終了しないので終了までの時間を定義するなどで終了する必要がある. 仕様記述では状態ごとに仕様が異なる場合に対応する必要がある. 複数の決定表を用意することで状態を持つパターンに対応できると考える. 自動生成の出力として Java のコードを生成したがそれ以外の言語でも自動生成できるようにツールの拡張が必要である.

参考文献

[1] D. S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing(OMG), Wiley, 2003

[2] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[3] IPA, ソフトウェア開発データ白書 2014-2015, IPA, 2014.

[4] 沢田 篤史, 平山 雅之 (編著), 組込みソフトウェア開発技術, CQ 出版.