社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2005－SE－150 (12)
2005／11／29

ステートマシン図を用いたプログラムの自動生成支援

平野雄一[†] 伊藤 恵[††]

近年，ソフトウェア開発は開発対象の分野にかかわらず，システムの大規模化と複雑化により難しいものとなってきている．そのような現状を解決するための一手法として，現在のプログラムコード中心のソフトウェアの記述に代わり，直感的に記述・理解のたやすい状態遷移図からプログラムコードを生成することを試みた．ここでは状態遷移図としてUML 2.0で定義されているステートマシン図を用い，ステートマシン図の状態（State）と遷移（Transition）にネイティブな実行コードを埋め込んでいくことで，ステートマシン図記述からネイティブなプログラムコードの生成を行っている．これによりシステムの問題領域を状態遷移図で表現される「制御」と，図上に埋め込まれる実行コードで表現される「動作」に分割することにより，ソフトウェア開発の効率化を試みている．評価のため，ステートマシン図からプログラムコードの生成を行うグループと，手動でプログラムコードを記述するグループに分け，比較実験を行い，今後の課題を検討する．

A Support of Automated Program Generation from Statemachines

Yuichi Hirano [†] and Kei Ito [††]

We propose an automated program generation from UML 2.0 statemachines while more efficient of the software developing. It generates executable native program codes from the statemachines which native codes is embedded in its states and transitions. It divides the problem area of the development software into the "control" on the statemachine and the "operation" on the native codes and efficient of the development.

1. はじめに

近年のソフトウェア開発は開発対象の分野にかかわらず，システムは大規模化・複雑化しており，そのような現状にかかわらずソフトウェアの開発期間は短納期化する傾向にある．ソフトウェア開発は難しいものとなってきており，同時に開発には膨大なコストを必要とするようになってきているため，ソフトウェア開発を効率化することによって開発にかかるコストを抑制することが必要である．

† 公立はこだて未来大学大学院 システム情報科学研究科
Graduate School of Systems Information Science, Future University-Hakodate
†† 公立はこだて未来大学大学 システム情報科学部 情報アーキテクチャ学科
Department of Media Architecture, School of Systems Information Science, Future University-Hakodate

ソフトウェア開発の効率化の手法として現在注目されているのが，OMGが提唱するMDA（Model Driven Architecture）[1]である．これはシステムの記述としてUMLなどのモデル記述を用い，モデル記述からプログラムコードを生成するなどして，モデルを実行可能なプログラミング言語として扱うことで，ソフトウェアの生産性と品質を改善する手法である．MDAの手法として，Mellorらが提唱するExecutable UML[2]や，Nucleus BridgePointやiUMLなどのMDAの対応をうたったソフトウェアがすでにいくつか登場しているが，現在のところMDAが普及してきたとはいえない状況である．この理由として，MDAを容易に試す環境が無く，開発者の間に浸透していないことや，MDAで主に使われるUMLは図の種類がUML 2.0の場合で14種類と多く，記法も多岐にわたるために習得が容易ではないこと，MDAの場合は特にモ

デル記述に厳密性が求められるためにモデル間の整合性の検証が難しいことなどがあげられる．前述したBridgePoint や iUML についても，1 つの対象のコードを生成するために複数の図を記述する必要があり，モデル間の整合性を意識しながらモデルを記述するのは容易ではない．また，組み込み分野においては，UML が本来用いられてきたビジネス系システムではデータ処理が中心であることに対して，組み込みシステムでは制御が中心となるために UML がなじみにくいということがあげられる．

組み込みシステムへの UML の適応に関しては，Douglass がリアルタイムシステムについての UML の適応について述べている[3] ほか，渡辺らが組み込みへの UML 適応について実践原則をまとめた eUML を提案している[4] が，これらは組み込みシステムへの UML 適応について述べるにとどまっており，依然としてプログラムコード中心の開発である．小澤らは UML における状態遷移表からのソースコード変換を提案している[5] が，ソースコードのスケルトンの生成にとどまっている．

これらに対し，ソフトウェア開発の効率化という問題について，直感的に記述・理解のたやすい状態遷移図のみを用い，状態遷移図に実行コードを埋め込み，それから実行可能なプログラムコードを生成することを提案する．本研究では，状態遷移図として UML 2.0[6] で定義されているステートマシン図を用い，ステートマシン図の状態と遷移にネイティブな実行コードを埋め込んでいくことで，ステートマシン図記述からネイティブなプログラムコードの生成を試みた．

2. ステートマシン図からのプログラム自動生成

2.1 環　　境

本研究では，ステートマシン図の記述のために CASE ツールである Poseidon for UML 3.1 Community Edition[7] を用い，ステートマシン図の記述から小型ロボット Khepera（**図 1**）を動作させるための Java 言語のコード生成を行った．

Khepera はスイスの K-Team 社が開発した小型二輪ロボットで，シリアル通信で制御可能であり，光センサと接近センサをそれぞれ前面 6 個と後方 2 個を備えている（**図 2**）．

2.2 ステートマシン図への Java コードの埋め込み

ステートマシン図からプログラムコードの自動生成を行う手法として，ステートマシン図の状態に動作実行のための Java 言語のコードを埋め込み，遷移に条件判断のための Java 言語のコードを埋め込む．状態に埋め込むことのできるコードは，ステートマシン図の仕様に従い，状態到達時に実行する Entry，一つの状態にとどまっている間実行し続ける DoActivity，次の状態に遷移する時に実行する Exit の 3 種類とする．遷移に埋め込むことのできるコードは，遷移条件を記述する Guard と，遷移後に実行する Effect の二種類とする．本来ならば遷移には Event を記述する必要があるが，本研究ではステートマシン図が実行の最上段におかれているために，遷移をすべて遷移条件で記述しうるので，Event を用いずに Guard を用いている．状態と遷移に埋め込むコードで使用する変数とオブジェクトのインスタンスの定義は，初期状態から出る遷移の Effect に記述する．記述された変数とインスタンスは，ステートマシン図上の同じ階層と下位の階層で使用することができる．

図 3 のように記述される状態（Simple State）の場合，状態名は “ForwardKhepera”，状態 “Forward-Khepera” 到達時に実行するコードは

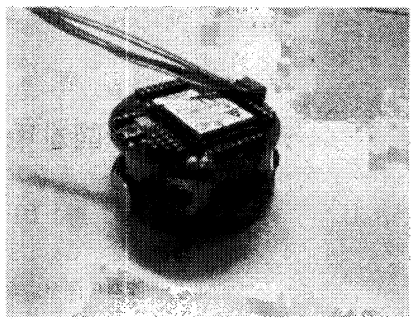

図 1 Khepera
Fig. 1 Khepera

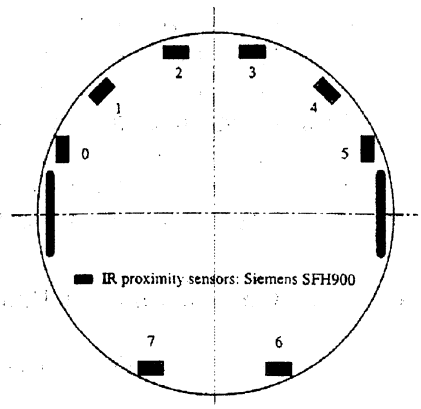

図 2 Khepera のセンサの位置[8]
Fig. 2 Position of the Khepera sensors[8]

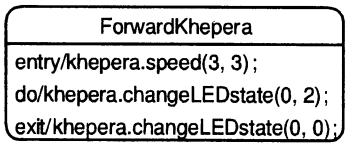

図 3 状態の例
Fig. 3 The example of the state

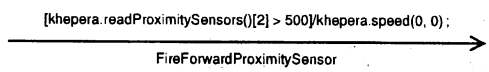

図 4 遷移の例
Fig. 4 The example of the transition

```
khepera.speed(3, 3);
```

(Khepera を速さ 3 で前進させる命令.), 状態 "ForwardKhepera" にとどまっている間実行するコードは

```
khepera.changeLEDstate(0, 2);
```

(Khepera の LED0 番の状態を切り替える命令. 実行時に LED 点灯の場合は消灯, 消灯の場合は点灯する.), 状態 "ForwardKhepera" から次の状態に遷移する時に実行するコードは

```
khepera.changeLEDstate(0, 0);
```

(Khepera の LED0 番を消灯させる命令.) となる.

図 4 のように記述される遷移 (Transition) の場合, 遷移名は "FireForwardProximitySensor", 遷移条件は

```
khepera.readProximitySensors()[2] > 500
```

(Khepera の接近センサ 2 番の値が 500 以上だった場合に true となる条件判断文.), 遷移後に実行するコードは

```
khepera.speed(0, 0);
```

となる.

図 5 のように記述される開始状態 (Initial Pseudo State) と遷移の場合, 開始状態名は "Init", 遷移名は "InitTrans" となり, 遷移の Effect として記述されたオブジェクトのインスタンス "khepera" と変数 "counter" がステートマシン図上で使用することができる.

本研究では, 状態と遷移のほかにも,

- 終了状態 (Final State)
- コンポジット状態 (Composite State)

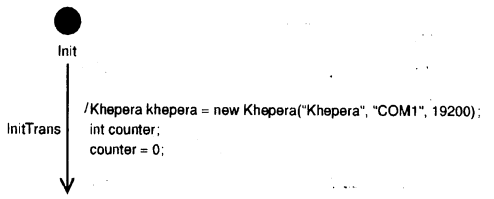

図 5 開始状態から出る遷移の例
Fig. 5 The example of the initial pseudo state and transition

- 並行状態 (Orthogonal State)
- 選択 (Choice Pseudo State)

を用いた.

2.3 ステートマシン図の記述

ステートマシン図の記述例として, **図 6** に, Khepera が前面の 6 個の光センサを使い, 光源を追跡する動作を記述したステートマシン図を示す.

初期状態 "Init" から伸びる遷移 "InitTrans" の Effect に, ステートマシン図上で使用する変数とオブジェクトのインスタンスの定義, プログラムの起動時に実行するコードを記述する. ここで定義した変数とインスタンスは, 同じ階層内の状態と遷移に埋め込む実行コードで用いることができる.

初期状態から状態 "Forward" に遷移し, 状態 "Forward" に到達すると Khepera を前進させるためのコードを実行し, Khepera 前面左側の光センサ (図 2 中の 0 もしくは 1) がセンサの中で最も明るい場合には状態 "TurnLeft" へ遷移 (F2L) し, 前面右側の光センサ (図 2 中の 4 もしくは 5) が最も明るい場合には状態 "TurnRight" へ遷移 (F2R) する. 状態 "TurnLeft" に到達すると Khepera を左回転するコードを実行し, 状態 "TurnRight" に到達すると右回転するコードを実行する. 状態 "TurnLeft" は, Khepera の前面前側の光センサ (図 2 中の 3 もしくは 4) が最も明るい場合には状態 "Forward" へ遷移 (L2F) し, 前面右側の光センサが最も明るい場合には状態 "TurnRight" へ遷移 (L2R) し, 状態 "TurnRight" は, Khepera の前面前側の光センサが最も明るい場合には状態 "Forward" へ遷移 (R2F) し, 前面左側の光センサが最も明るい場合には状態 "TurnLeft" へ遷移 (R2L) する.

2.4 ステートマシン図から Java 言語への変換

ステートマシン図から Java への変換は, Gamma らが提案しているデザインパターン[9] の 1 つである State パターンを基にした構造を定義し, その構造に従い行った.

複数の状態を統括する StateRoot クラスと, 状態のインターフェースである IState クラス, 状態ごと

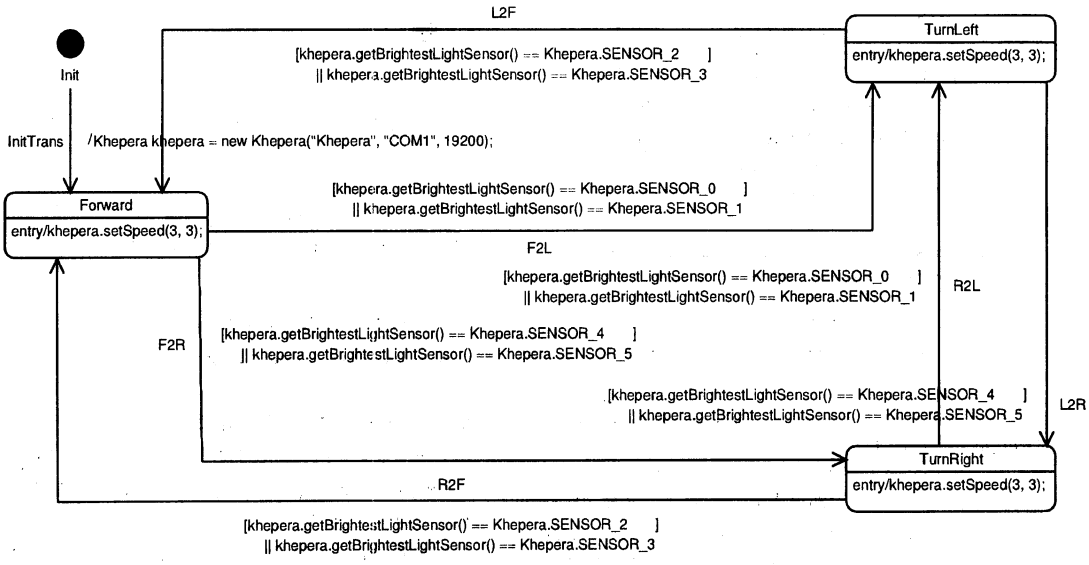

図 6 Khepera が光源を追跡するためのステートマシン図
Fig. 6 The staternachine diagram for the khepera track a light source

に SimpleState クラスを生成し，SimpleState クラス内に状態内の Entry/doActive/Exit と，状態から伸びる遷移の遷移条件判定と到達先状態を記述する形にした．生成されるプログラムコードのクラス図の概要は**図 7** のようになる．

コンポジット状態または並行状態の場合は，CompositStateChild クラスを生成し，その下に並列状態の数の StateRootThread クラスを生成，StateRootThread 下の状態のインターフェースである IState クラスと各状態のクラスを生成する．CompositStateChild クラスから，各 StateRootThread クラスを別スレッドとして起動している．生成されるプログラムコードのクラス図の概要は**図 8** のようになる．コンポジット状態または並行状態が入れ子になっている場合にも対応している．

3. 結　　果

上述のシステムを Java Standard Edition 5.0 と DOM (Document Object Model) を用いて実装した．システムの入力は，CASE ツールが出力する XMI (XML Metadata Interchange) [10]，出力は実行可能な Java のコードである．

図 6 で示したステートマシン図を XMI 形式で出力し，システムに XMI を入力すると、リスト 1・2・3☆のようなコードが生成された．生成されたコードを実行すると，実際に Khepera が光源を追跡した．

```
package jp.ac.fun.khepera.generatedcode.go2light;

import jp.ac.fun.khepera.lib.Khepera;

/**
 * StateMachine KheperaDiagram class.
 * Generate 2005/10/17 12:50:37 JST.
 *
 * @author Deuce Khepera Code Generator
 * (GeneratorVersion 2005/9/15.
 * ParserVersion 2005/9/7.)
 */
public class StateRootKheperaDiagramTopReg {
  IStateKheperaDiagramTopReg state;

  public StateRootKheperaDiagramTopReg() {
    Khepera khepera = new Khepera("Khepera", "
      COM1", 19200);

    /* 初期の状態 */
    state = new
      SimpleStateChildKheperaDiagramTopRegForward(
      khepera);

    while (state != null) {
      state = state.drive();
    }
  }
```

☆ リスト 3 は状態 "Forward" のコードを生成したもの．状態 "TurnLeft"，状態 "TurnRight" それぞれに同様のコードが出力される．

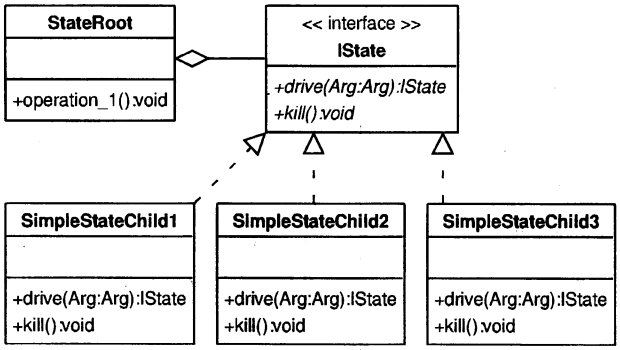

図 7　生成されるコードのクラス図

Fig. 7　The class diagram of the generated codes

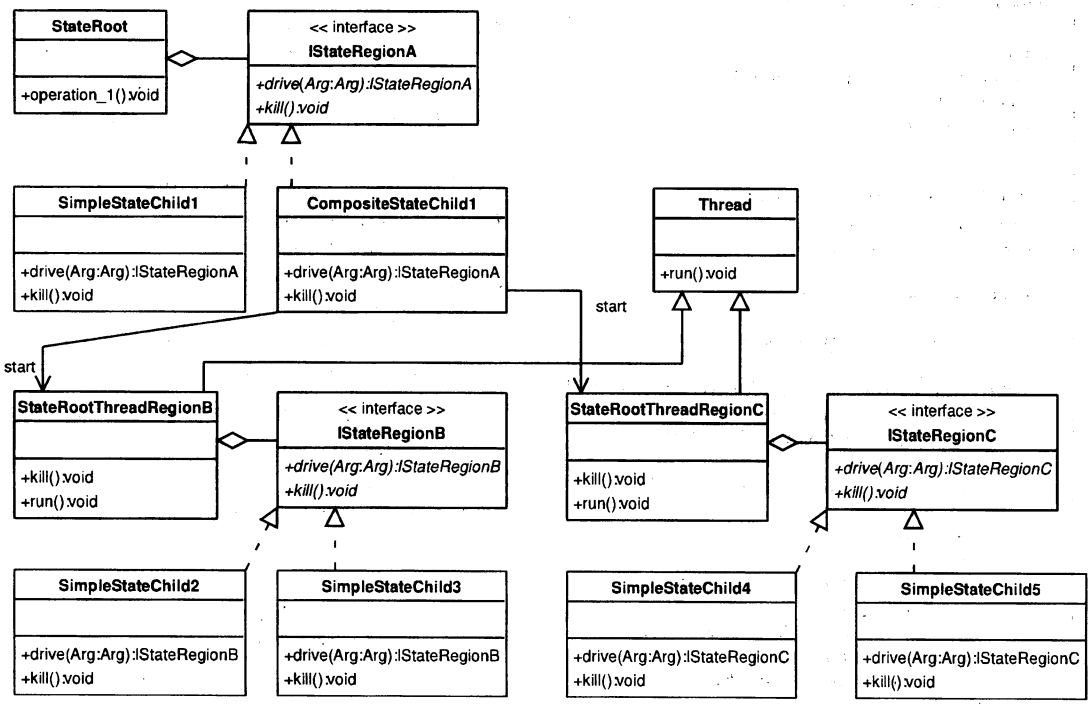

図 8　生成されるコードのクラス図（複数の階層が存在する場合）

Fig. 8　The class diagram of the generated codes (contain some regions)

```
  public static void main(String[] args) {
    new StateRootKheperaDiagramTopReg();
  }
}
```

リスト 1　StateRoot

```
package jp.ac.fun.khepera.generatedcode.go2light;

/**
 * State interface.
 * Generate 2005/10/17 12:50:37 JST.
 *
 * @author Deuce Khepera Code Generator
 * (GeneratorVersion 2005/9/15.
 * ParserVersion 2005/9/7.)
 */
public interface IStateKheperaDiagramTopReg {
  public IStateKheperaDiagramTopReg drive();

  public void kill();
}
```

リスト 2　IState

```
package jp.ac.fun.khepera.generatedcode.go2light;

import jp.ac.fun.khepera.lib.Khepera;

/**
 * Vertex Forward class.
 * Generate 2005/10/17 12:50:37 JST.
 *
 * @author Deuce Khepera Code Generator
 * (GeneratorVersion 2005/9/15.
 * ParserVersion 2005/9/7.)
 */
public class
  SimpleStateChildKheperaDiagramTopRegForward
  implements IStateKheperaDiagramTopReg {
  private Khepera khepera;
  private boolean running = true;

  public
    SimpleStateChildKheperaDiagramTopRegForward(
    Khepera khepera) {
    this.khepera = khepera;
  }

  public IStateKheperaDiagramTopReg drive() {
    /* 状態Forward の Entry の動作 */
    khepera.setSpeed(3, 3);

    while (running == true) {

      /* 状態Forward の doActive の動作（この例では空）*/

      if ((khepera.getBrightestLightSensor() == Khepera
        .SENSOR_0) || (khepera.getBrightestLightSensor
        () == Khepera.SENSOR_1)) { /* 遷移条件 */

        /* 状態Forward の Exit の動作（この例では空）*/

        /* 遷移先 */
        return new
          SimpleStateChildKheperaDiagramTopReg
          TurnLeft(khepera);
      } else if ((khepera.getBrightestLightSensor() ==
        Khepera.SENSOR_4) || (khepera.
        getBrightestLightSensor() == Khepera.
        SENSOR_5)) { /* 遷移条件 */

        /* 状態Forward の Exit の動作（この例では空）*/

        /* 遷移先 */
        return new
          SimpleStateChildKheperaDiagramTopReg
          TurnRight(khepera);
      }
    }

    return null;
  }

  public void kill() {
    running = false;
  }
}
```

リスト 3　State

この場合，ステートマシン図の状態数は 3 となるため，生成されるソースコードのクラス数は StateRoot が 1 クラス（階層につき 1 クラス），IState が 1 クラス（階層につき 1 クラス），SimpleStateChild が 3 クラス（状態数と同数）となり，あわせて 5 つのクラスとなる．

4. 実　　験

4.1 実験方法

被験者を，Java コードを手動で書くグループと，ステートマシン図を描いてステートマシン図からコードを生成するグループの 2 グループに分け，Khepera を動作させるためのコードを書いてもらい，比較実験を行った．

それぞれのグループに簡単な練習課題を 3 題を行ってもらい，Khepera の動作方法と，ステートマシン図からコードを生成するグループにはステートマシン図からのコードの生成の方法を学んでもらった．その後，それぞれのグループに

(1) 光センサ 0 もしくは 1 が光センサのなかで最も明るいときには，LED の 1 番を消灯，LED の 0 番を 1 秒間隔で点滅させながら，Khepera を速さ 3 で左旋回させる．

(2) 光センサ 2 もしくは 3 が光センサのなかで最も明るいときには，LED の 0 番と LED の 1 番を 1 秒間隔で点滅させながら，Khepera を速さ 3 で前進させる．LED の 0 番と LED の 1 番の点滅の同期は取らなくてもよい．

(3) 光センサ 4 もしくは 5 が光センサのなかで最も明るいときには，LED の 0 番を消灯，LED の 1 番を 1 秒間隔で点滅させながら，Khepera を速さ 3 で右旋回させる．

(4) 上記の Khepera の動作開始から 60 秒後に Khepera を停止させ，LED の 0 番と 1 番を消灯させる．

の動作を行うコードを実装してもらった．

4.2 実験結果

今回の実験では，2 グループの間には実装に要した時間には大きな差が見られなかった．Java コードを

手動で書くグループは，被験者ごとにコードの構造に大きな差が見受けられたが，ステートマシン図からコードを生成するグループは大きな差は見受けられなかった．

Java コードを手動で書くグループは，

- スレッドごとの内容の記述は簡単だった．
- マルチスレッドの扱いが難しかった．

という意見が目立った．ステートマシン図からコードを生成するグループは，

- ステートマシン図の記述に最初はとまどったが，練習課題をこなすうちに理解できた．
- 考えていたよりも直感的に書けた．
- スレッド処理を意識せずに書けた．
- ステートマシン図の記述に若干手間がかかった．
- モデリングツールが使いにくい部分があった．

という意見が目立った．

5. 考　　察

5.1 開発効率・長所

ステートマシン図を CASE ツールを用い GUI での表記が可能になったことで，コードを手動で記述する場合に比べ直感的に理解が可能となりそうである．これにより，開発者が意図しない動作を記述してしまうことを防ぐ効果も期待される．また，ステートマシン図の動作をソフトウェア上でシミュレーションする環境をあわせて用いることで，ステートマシン図の記述の矛盾やミスを発見しやすくなることもあわせて期待される．

本システムではステートマシン図上にネイティブな実行コードを埋め込む手法を採っているが，実行コードを別の XML ファイルに記述し，ステートマシン図の各エレメントに名前をつけて，名前と実行コードとを対応づけしていき，プログラムコードを出力するシステムをすでに試作している．これを使用した場合，ステートマシン図上に実行コードを埋め込む必要が無く，あらかじめ実行コードを記述した XML ファイルを用意すれば，ステートマシン図記述者は実行コードを意識することなくステートマシン図を記述することが可能になる．これにより，よく使われるコードをあらかじめ記述しておくことで，記述の省力化が行えることや，ステートマシン図記述者がプログラミング技術を習得している必要が無くてもよいことが期待されるほか，実装コードを記述した XML を差し替えることで，同一のステートマシン図からほかの機器のプログラムコードを出力することができるようになることが期待される．

5.2 開発効率・短所

生成されるコードの可読性が手動でコードを記述した場合に比べて低くなることが懸念される．状態数に応じてクラスを生成するために，状態数が大きなステートマシン図の場合にはクラスも多くなることから，コードとして読みにくく，内容を追いにくくなるために，生成したコードに手動で修正を加える場面が発生した場合には，コードの修正を行いにくくなることが予想される．これについては，現在，単一の Java パッケージとしてコードを生成しているが，これをステートマシン図上の階層ごとにパッケージを分割し生成することや，クラスの命名をわかりやすくするといった手法が考えられる．

また，生成されるコードそのものも手動で記述した場合に比べて効率が低く，冗長なコードを生成する傾向にある．今回はプログラムを PC の上で実行し Khepera をシリアル通信で制御しているために，プログラムの効率や大きさが致命的な問題になる場面は無かったが，これをほかの環境，特に実時間の制約に厳しく，リソースが少ない組み込みの環境などに適応しようとした場合には，生成されるコードの効率と大きさは大きな問題になるであろう．これについては，Niaz らの提案する UML ステートチャート図と Java コードのマッピング手法[11),12)] を参考に，効率の改善を検討している．

モデリングツールそのものの出来が開発効率に大きな影響を与えることが懸念される．ステートマシン図記述のためのモデリングツールとして Poseidon for UML 3.1 を用いたが，Poseidon は Pure Java で書かれているために重く，ステートマシン図を記述することにストレスを感じるといったことや，図にコードを埋め込むために何回もクリックしなければメニューに到達できないなどのモデリングツールそのものへの不満が実験で聞かれた．今後，扱いやすいモデリングツールの出現が期待される．

5.3 スレッド実装の隠蔽

本システムを用い，ステートマシン図上でコンポジット状態もしくは並行状態を記述した場合，スレッドを用いたコードを生成する（図 8）．コードを生成することで，スレッド処理の実装はステートマシン図上のコンポジット状態もしくは並行状態といった形で隠蔽された形となるが，これにより，一般的に難しいとされるスレッド処理を手動で記述するよりも，コードを生成することでスレッド処理を用いた実装の記述効率向上が期待される．

5.4 モデルと実装の格差の解消

Rumbaughは，オブジェクト指向開発においてモデルとシステム実装には格差があることを指摘している[13]が，オブジェクトの相互作用のみでシステムを記述するのが難しいことがその一因である．

本研究では，ステートマシン図が埋め込んだ実行コードのトリガーの役割を果たすことで，埋め込まれた実行コードのオブジェクト（モデル）とステートマシン図で記述されるコードの実行条件（システム実装）を問題領域として切り分けることができ，ステートマシン図から後者は自動的に生成される．モデル上の問題をオブジェクト指向で記述し，システム実装上の問題をステートマシン図で記述することで，モデルとシステム実装の格差の解消を行えることが期待される．

6. 今後の方向性

製作したシステムは現在のところステートマシン図の仕様を十分に準拠したとはいえないために，今後さらに拡充していく予定である．また，Kheperaはセンサがあわせて2種16個と少ないために，ステートマシン図が小さくまとまってしまう傾向にあるが，これは今後ほかの対象を用いて，図が大きくなった場合に開発効率に対して本システムがどのような影響を与えるのかを検討，議論していく必要がある．

参 考 文 献

1) Franknel, D.S.，日本アイ・ビー・エム株式会社 TEC-J MDA 分科会：MDA™ モデル駆動アーキテクチャ，エスアイビー・アクセス (2003).
2) スティーブJ. メラー，マークJ. バルサー：*Executable UML*, 翔泳社 (2003).
3) ブルースダグラス：リアルタイム UML 第2版，翔泳社 (2001).
4) 渡辺博之，渡辺政彦，堀松和人，渡守武和記：組み込み UML eUML によるオブジェクト指向組み込みシステム開発，翔泳社 (2002).
5) 小澤陽平，細川卓誠，小泉寿男：組み込みソフトウェア向け UML における状態遷移表・ソースコード変換方式，FIT（情報科学技術フォーラム）2003，pp.165–166 (2003).
6) Object Management Group, Inc.: UML 2.0 Superstructure Specification, `http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf`.
7) Boger, D.M., Sturm, T., Schildhauer, E. and Graham, E.: *poseidon for uml user guide*, Gentleware AG (2004).
8) K-Team: *Khepera USER MANUAL*, K-Team, version 5.0 edition (1998).
9) エリックガンマ，ラルフジョンソン，リチャードヘルム，ジョンブリシディーズ：オブジェクト指向における再利用のためのデザインパターン，ソフトバンクパブリッシング (1999).
10) Object Management Group, Inc.: XML Metadata Interchange (XMI) Specification, `http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf`.
11) Niaz, I.A. and Tanaka, J.: Code Generation From UML Statecharts, *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003)*, Marina Del Rey, USA, pp.315–321 (2003).
12) Niaz, I. A. and Tanaka, J.: Mapping UML Statecharts To Java Code, *Proceedings of the IASTED International Conference on Software Engineering (SE 2004)*, Innsbruck, Austria, pp. 111–116 (2004).
13) Rumbaugh, J. E.: Modeling Through the Years., *JOOP*, Vol.10, No.4, pp.16–19 (1997).