2 組み込みソフトウェアの設計モデリング技術

鵜林 尚靖
九州工業大学 情報工学部 知能情報工学科
ubayashi@acm.org

組み込みソフトウェア開発では，1つの開発で複数の製品を開発するプロダクトラインの実現，ハードウェア要求を加味した効率的な実装，性能などの非機能要求の実現，高いレベルの品質確保，などが設計を進める上で課題となる．本稿では，これらの課題を解決するため過去どのような開発手法が提案されてきたか振り返るとともに，どのような問題が依然残っているのか見ていく．さらに，次世代の開発手法として，組み込みソフトウェアをモデル駆動で開発する方法をアスペクト指向や形式検証などの要素技術を取り込みながら説明する．

組み込みソフトウェア開発のポイント

組み込みソフトウェア開発と一言でいってもその範囲は広いが，一般的に次のような点が設計を進める上でポイントとなる．

プロダクトラインを考慮した設計：1つの開発で複数の製品を開発する場合が多い．このような開発形態をプロダクトライン型開発という[1)]．標準的な機能は同じであるが，機種ごとに一部の機能が追加になったり削除されたりする．また，機種間でプログラムの実行プラットフォームが変わる場合もある．標準的な機能と機種ごとに変動する機能を分けるSV（Standard/Variable）分離という考え方が重要となる．

ハードウェア要求を考慮した設計：組み込みソフトウェアはハードウェアと協調して初めてシステムとして動作する．そのため，応答時間，タイミング，CPUやメモリなどのリソース面での制約が大きい．また，フェールセーフやフェールソフトなどの対策も重要となる．

機能要求／非機能要求の反映：組み込みソフトウェアが利用者に提供する機能要求（functional requirements）以外に，前項で述べたような要求を開発しなければならない．このような要求は非機能要求（non-functional requirements）と呼ばれる．ただ，これをモジュール性よく実装するのは容易ではない．非機能要求の実装はプログラム中にさまざまな個所に散らばってしまいがちになるからである．

高度な品質：組み込みソフトウェアの機能は年々豊富になっており，開発規模は急激に膨らんでいる．また，機器が不特定多数に使用されるため，求められる品質も非常に高い．その一方で，ハードウェアと連動して動作するため，タイミングなどの正しさを保証するのは容易ではない．同じようにテストしても正常に動作したりしなかったりする場合が多く，通常，テストだけでバグを取り除くことは難しい．

これらは特定の手法には依存しない組み込みソフトウェアの設計という問題に共通する課題である．本稿では，これらの課題を解決するため過去どのような開発手法が提案されてきたか振り返るとともに，どのような問題が依然残っているのか見ていく．さらに，次世代の開発手法として，組み込みソフトウェアをモデル駆動で開発する方法をアスペクト指向[2)]や形式検証などの要素技術を取り込みながら説明する．新しい技術により，これからの組み込みソフトウェア開発がどのように変化していくのか感じ取っていただけたら幸いである．

組み込みソフトウェア開発手法の変遷

組み込みソフトウェアはハードウェアと協調して動作するという性格上，ハードウェアの機能や性能を最大限に活かすことが重要となる．そのため，**表-1**に示すように今までさまざまな開発手法が提案されてきた．ここ

時代区分	主な開発手法
第1期 構造化手法の時代（1980～90年代）	リアルタイムSA（Word&Mellor, Hatley&Pirbhai），DARTS/ADARTS/CODARTS（Gomaa）
第2期 オブジェクト指向の時代（1990年代～現在）	COMET（Gomaa），OCTOPUS（ノキア），ROPES（Douglass），ROOM（Selic他），Executable UML（Mellor他），eUML（渡辺他）
第3期 ポストオブジェクト指向の時代（今後）	モデル駆動開発，アスペクト指向，形式検証など

表-1　組み込みソフトウェア開発手法の変遷

では，便宜上，時代を，第1期（構造化手法の時代），第2期（オブジェクト指向の時代），第3期（ポストオブジェクト指向の時代）の3つに分類した．

●第1期 構造化手法の時代

従来，組み込みソフトウェア開発というと，ハードウェアの性能を最大限に引き出すためアセンブラでプログラムを作成するのが一般的であった．しかしながら，アセンブラでは開発生産性の向上に限界があるし保守も大変である．また，ハードウェアが変更になると今までのソフトウェア資産が役に立たなくなってしまう．そのような理由もあって，1980年代後半から，組み込みソフトウェアの開発にもC言語などの高級言語が採用されることが多くなった．第1期（1980～90年代）ではアセンブラやC言語などでプログラミングが行われたため，その上流の分析や設計にはリアルタイム向けの構造化手法が用いられることが多かった．どのようなタイミングでどのようにハードウェアを制御するかを記述するために状態遷移図を利用したり，モジュール分割の指針としてタスク分割を適用したりしていた．この時代の代表的な手法として，Word&MellorやHatley&PirbhaiらのリアルタイムSA（Structured Analysis），GomaaらのDARTS（Design Approach for Real-Time Systems）およびその派生版であるADARTS（Ada-based DARTS）/CODARTS（COncurrent DARTS）などがある．

●第2期 オブジェクト指向の時代

1990年代中頃からソフトウェア開発の世界ではオブジェクト指向技術が注目され，現在に至っている．構造化手法と比較してモジュール性や再利用性に優れているというのがその主な理由である．UML（Unified Modeling Language）[3]やJavaが急速に広まり，今後，開発にオブジェクト指向を取り入れる傾向はますます強くなると思われる．組み込みソフトウェアの世界でもオブジェクト指向を取り入れようという試みが数多くなされてきた[4]．代表的な手法として，Gomaaが提唱するCOMET（Concurrent Object Modeling and architectural design mEThod），ノキアで開発されたOCTOPUS，DouglassのROPES（Rapid Object-Oriented Process for Embedded System），SelicらのROOM（Real-time Object-Oriented Modeling），MellorらのExecutable UML[5]，渡辺らのeUML（embedded UML）[6]などがある．開発支援ツールの面では，RoseやStatemateなどのモデリングツール，KISS, Rose RT, BridgePoint, Rhapsodyなどの組み込み向けツールが提供されている．

それでは，オブジェクト指向による組み込みソフトウェアの開発スタイルがどのようなものか簡単に見ていこう．

オブジェクト指向による分析，設計

ハードウェア機器に組み込むソフトウェアを開発するには，最終的にプログラムコードの形に落とし込まなければならないが，通常いきなりプログラミングから開始することはない．組み込みソフトウェアが動作する環境，提供する機能，性能要求などを分析した上で，それをどう実現するかという観点から設計を行う．その後，設計からプログラムコードを作成し，最後にテストを行う．すなわち，分析，設計というフェーズを経て，初めてプログラミングを行うのが一般的な開発スタイルである．この際，分析，設計のフェーズでモデルを作成する．モデルとは対象の本質的な特徴を抽象化したものであり，分析モデルでは何（what）の構造を，設計モデルでは手段（how）の構造を，記述する．組み込みソフトウェアの場合，ハードウェアと協調してどうリアクティブに振る舞うべきか，という動的な視点からモデルを作成することが重要である．

オブジェクト指向によるモデルとはどのようなものであろうか？　一例を**図-1**に示す．この図はディジタル時計の振る舞いを示したもので，UMLのクラス図とそれに対応するステートマシン図が記載されている．この図はステートマシンがクラス中の操作とどのように関連しているかを示したものである．このクラスでは，時刻表示（Display），時間設定中（Set hours），分設定中（Set minutes）の3つの状態を持つ．mode_button操作は3つの状態間を遷移させる機能を，inc操作は時間あるいは分の値を1つだけ増分させる機能を，compWorldTime操作は世界各地の時刻を計算する機能を持つ．このような

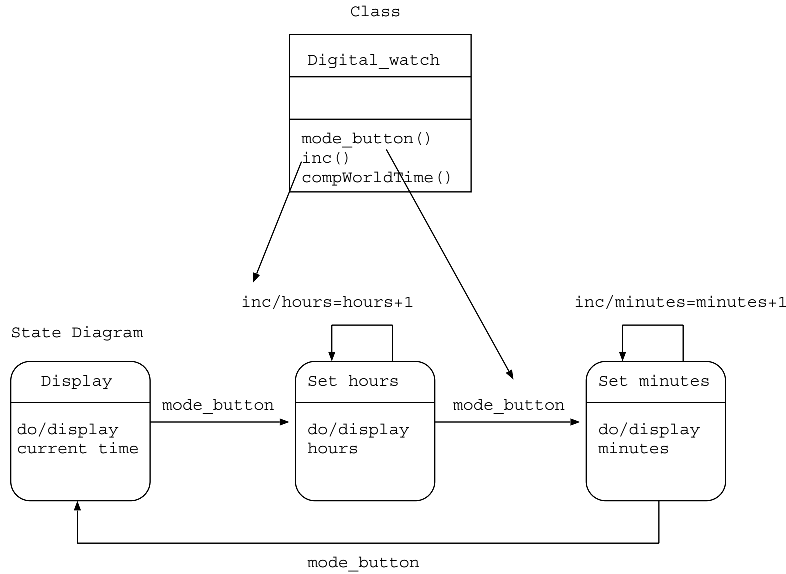

図-1　UMLによるディジタル時計のモデル化（文献7）より引用したものを若干修正）

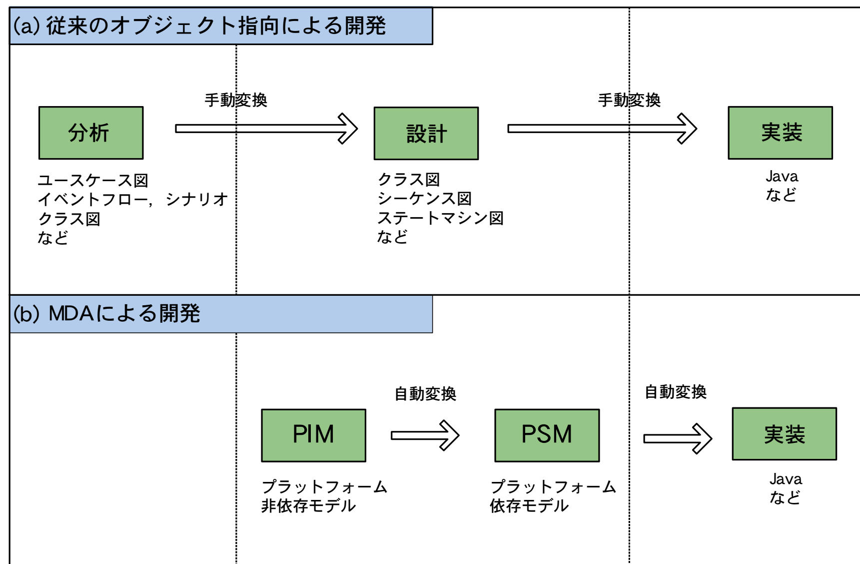

図-2　オブジェクト指向開発とMDAの対比

モデルを作成することにより，ディジタル時計の振る舞いが明確になる．

UMLのステートマシン図は構造化手法における状態遷移図に対応する．状態遷移図は構造化手法の中でも重要な役割を果たしていたが，オブジェクト指向でもその重要性は変わらない．オブジェクト指向の導入により構造化手法の考え方が捨てられたのではなく，振る舞いの記述やハードウェア特性の記述など重要な部分はちゃんと生き残っている．むしろ，構造化手法からオブジェクト指向に発展したと捉える方が正確である．

オブジェクト指向による開発手順

オブジェクト指向による開発手順は大まかには以下のようになる（**図-2（a）**）．

分析フェーズ：組み込みソフトウェアに求められる機能要求を分析し，それを分析モデルとしてまとめる．UMLのユースケース図やクラス図などで表現する．非機能要求も洗い出す．

設計フェーズ：分析フェーズで整理した要求をシステムとしてどう実現するかという観点から設計モデルを作成する．特に振る舞いの観点から，分析フェーズのクラス図をブレークダウンするとともに，シーケンス図やステートマシン図を作成する．その際，物理時間，タイミング仕様，リソース，並行性とスケジューリン

グ，などを明確にする．この場合，リアルタイム向けのUML プロファイル（UML Profile for Schedulabilty, Performance, and Time）などが役に立つ．なお，UML の新しいバージョンであるUML2[3] からは組み込みソフトウェアの開発に有効なコンポジット図やタイミング図などが追加になっている．

実装フェーズ：設計モデルから実行プラットフォームに対応したプログラムコードを作成する．単純に設計モデルをプログラムコードに変換するだけでなく，タイミングやリソース制約などの非機能要求を満たすように最適化しなければならない．

●第3期 ポストオブジェクト指向の時代

組み込みソフトウェアの開発にオブジェクト指向を導入することにより，構造化手法よりも分かりやすいモデルを構築することが可能になる．しかし，本稿の冒頭で提示した課題はどの程度解消されたのであろうか？

プロダクトラインを考慮した設計：分析や設計時のモデル資産を蓄積することにより，ある程度のプロダクトラインを構築することができる．しかしながら，オブジェクト指向による設計モデルの多くは実装に依存する部分と依存しない部分が明確に切り分けられていない場合が多く，モデルの再利用は限定的である．また，分析モデルから設計モデルへの変換，設計モデルからプログラムコードへの変換は多くの場合人手で行われている．せっかくモデルを作成しても，直接プログラムコードにはつながらないという問題が残されている．

ハードウェア要求を考慮した設計，
機能要求／非機能要求の反映：UMLプロファイルを用いることにより，ハードウェア特性などの非機能要求の記述が可能である．しかしながら，まだ十分とはいえない．オブジェクト指向を導入しても，非機能要求の多くは，1つのモジュールにカプセル化できず，複数のモジュールにまたがってしまうからである．そして最も難しいのは，非機能要求を記述したモデルからそれを満たすプログラムコードを作ることである．たとえば，非機能要求として応答時間を指定した場合に，その応答時間内で処理が完了するプログラムコードを作成する必要がある．

高度な品質：オブジェクト指向を導入しても，残念ながら検証にかかわる問題は従来とあまり変化していない．

本稿では，上記の課題解決に有効だと考えられる次世代技術として，モデル駆動開発，アスペクト指向，形式検証の3つを取り上げる．次章以降，組み込みソフトウェアをモデルベースで開発する場合に，これらの技術がどのように役立つか例を交えながら説明する．

モデル駆動による組み込みソフトウェアの開発

●モデル駆動開発とは

モデル駆動開発手法で代表的なのがOMG（Object Management Group）で仕様策定が行われているMDA（Model-Driven Architecture）[8] である．MDAによる開発と従来のオブジェクト指向開発の違いは，主に設計フェーズにある．MDAでは，設計モデルを特定のプラットフォームや実装技術に依存しないPIM（Platform Independent Model）と，依存するPSM（Platform Specific Model）に分ける（図-2（b））．そして，PIMからPSMへはモデルコンパイラを用いて自動変換する．さらにPSMから特定のプログラミング言語に変換する．

図-1で示したディジタル時計のモデルはPIMに相当し，モデルコンパイラはこれを特定のプラットフォームで実行可能なPSM，さらにはJava等のソースコードに変換する．このような方式を採用することにより，以下のようなメリットが生まれる．

- 開発者は特定のプラットフォームやプログラミング技術にとらわれることなく，PIMの開発に全力を注ぐことができる．すなわち，従来のコーディング中心の開発からモデル中心の開発にパラダイムシフトすることが可能になる．

- 同じPIMから複数のPSMを生成することができる．すなわち，PIMモデル部品とモデル変換規則をライブラリ化することにより，さまざまな機能やプラットフォームに対応したプロダクト群を生成することが可能になり，プロダクトライン型開発の実現につながる．

●厳密なモデル表記とモデル変換定義

モデル駆動開発を実現するには以下の課題を克服することが鍵となる．

(a) **厳密なモデル表記**：モデルが厳密に書ける必要がある．そうでないと，モデルから実行可能なプログラムコードを生成できない．

(b) **厳密なモデル変換定義**：モデル変換規則を記述し，そ

算術演算子	+, -, *, /, =, <, >, <=, >=, <>
論理演算子	and, or, xor, not, implies, if/then/else
プロパティ演算子	.
コレクション演算子	collection- > size ( ) : integer collection- > forAll (x ｜ f (x)) : Boolean collection- > select (x ｜ f (x)) : collection collection- > exists (x ｜ f (x)): Boolean

表-2　OCLの主な演算子

れを部品化するための言語が必要となる．モデル変換部品を差し替えることにより，同じモデルから用途別に異なるプログラムコードを生成することが可能になる．

(a) については，UML2からメタモデルに基づいた厳密なモデル定義が可能になっている．また，モデル内容の厳密性という面からはOCL (Object Constraint Language)[9]が重要となる．(b) については，現在，MDAの一環としてOMGでQVT (Queries, Views, and Transformations)[10]というモデル変換言語が検討されている．

OCL

OCLはUMLモデルの整合性をモデル要素間で成立すべき制約条件として記述する言語である．具体的には，**表-2**に示すように，算術演算子，論理演算子，プロパティ演算子，コレクション演算子などを用いて制約条件を記述する．

図-1のディジタル時計の例で考えてみよう．このモデルは厳密なモデルといえるであろうか？ inc操作では単純に時間 (hours) または分 (minutes) の現在値を1つ増やしているだけであるが，時間が23時だと24時になってしまう．時間は0時から23時までで，23時の次は0時に戻さなければならない．OCLを用いると，以下のような厳密な記述が可能になる (ただし，0時から22時までの場合についてのみ記述)．ここでは制約条件を事前条件 (pre) と事後条件 (post) によって記述している．@preは操作実行前の値を示す．

```
context Digital_watch::inc()
pre: status = SetHours
     and 0 <= hours and hours < 23
post: hours = hours@pre + 1
```

QVT

MDAによる自動変換を可能にするにはモデル記述を厳密にするだけでは不十分である．モデル変換規則を厳密に記述し，また記述したものが再利用できなくてはならない．QVTはこのための言語で，問合せ (Queries)，ビュー (Views)，変換 (Transformations) の3つから構成される．問合せはモデルから特定の要素を選択する機能であり，QVTでは先に示したOCLの拡張版を問合せ言語として採用する予定である．ビューはモデルをある側面から切り出す機能，変換はあるモデルを更新したりそれから別の新しいモデルを生成する機能である．変換には関係 (relation) とマッピング (mapping) の2種類があり，前者は双方向の変換を，後者は単方向の変換を指す．QVTではさまざまなモデル変換を記述できる．MDAにおけるPIMからPSMへの変換規則もQVTを用いて記述できる．

以下は，UMLクラス (SM.Class) をJavaクラス (JM.Class) に単純に変換する規則をQVTにより記述した例である (文献10) より引用)．UMLのクラス名 (name) はそのままJavaのクラス名に変換され，UMLの属性 (attributes) は別の変換規則Simple_Attribute_To_Java_Attributeを用いてJavaの属性に変換される．この規則を用いることにより，図-1に示したディジタル時計のUMLモデル (PIM) からJavaクラス (PSM) を生成することが可能になる．

```
mapping Simple_Class_To_Java_Class
 refine Simple_Class_And_Java_Class {
 domain{(SM.Class)[name=n,attributes=A]}
 body {
  (JM.Class)[
    name=n,
    attributes = A->iterate(a as ={} |
     as +
      Simple_Attribute_To_Java_Attribute(a))
    ]
   }
 }
```

UMLのダイアグラムについては，このような変換規則を順次用意していけばよいが，先に述べたOCLについてはどのような変換が必要となるであろうか？ ディジタル時計の例では事前条件と事後条件をOCLで記述したが，この場合は以下のような言明 (assertion) を含むJavaメソッドに変換する規則を記述すればよい．

```
int inc(){
  assert status==SetHours
       && 0 <= hours && hours < 23;
  oldHours = hours;
```

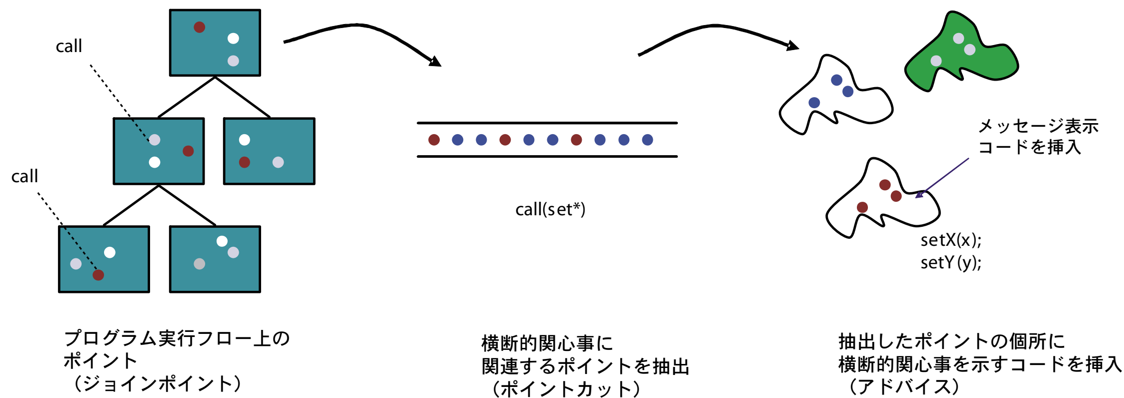

図-3　ジョインポイントモデル（文献 13）の図を一部変更）

```
  hours = hours + 1; //PIM に記載されたコード

  assert hours==oldHours+1;
}
```

アスペクト指向による組み込みソフトウェアの開発

●組み込みソフトウェアと非機能要求

MDAによりプラットフォームや実装技術に依存しないかたちでモデル中心の開発が実現できれば，組み込みソフトウェア開発の生産性は大幅に向上するであろう．また，プロダクトライン型開発への移行も現実味を帯びてくる．実は，現在でも，組み込みソフトウェア開発の世界では，ステートマシン図ベースでモデルを作成し，それからプログラムコードを生成するCASE（Computer Aided Software Engineering）ツールが存在する．これらのツールは必ずしもMDAツールと呼ばれていないが，MDAの考え方に近いものがある．ただ，現状でもこのようなCASEツールが存在しながら，必ずしも広く普及しているとはいえない．なぜであろうか？　組み込みソフトウェアの開発ではタイミングやリソース制約などの非機能要求の実装が重要となるが，現状のMDAでは以下のような問題が存在する．

- 非機能要求を記述するための表現手段が弱い．リアルタイム向けのUMLプロファイルは存在するが，非機能要求のすべてが記述できるわけではない. たとえば，複数のモジュールを横断するようなエラー処理などの記述はUMLプロファイルの範囲外である．
- 非機能要求をモデルとして記述するための手段が仮に提供されたとしても，その要求を保存したままプログラムコードを生成する手段がない. たとえば, モジュール性に優れたモデルが開発でき，それからプログラムコードが生成されたとしても，性能などの非機能要求を満たさなければ，生成されたコードをチューンアップしなければならない．通常チューンアップはプログラムコード中のさまざまな個所に影響を及ぼすことが多く，結局は分かりづらいコードをメンテナンスせざるを得なくなる．

組み込みソフトウェアに見られるこのような性質は，横断的関心事（crosscutting concerns）と呼ばれる．横断的関心事の存在は設計をやり難くしてしまう可能性がある[11]．このような問題を解決するための技術の1つがアスペクト指向である．アスペクト指向では横断的関心事をアスペクトと呼ばれるモジュールによって記述する．ここでは，アスペクト指向の概要，MDAへの応用方法について説明する．

●アスペクト指向

アスペクト指向は横断的関心事をモジュール化するための技術として，ここ数年，大きく注目されている．AspectJ[12] など実用的なプログラミング言語が提供されているのも普及を後押ししている．アスペクト指向のメカニズムはジョインポイントモデル（Join Point Model, 以下JPM）によって表現される．ここでは，AspectJの用語を用いてJPMについて簡単に説明する．

図-3に示すように，JPMは，ジョインポイント（join point），ポイントカット（pointcut），アドバイス（advice）の3つから構成される．ジョインポイントとは，プログラム実行フロー中のポイントのことである．たとえば，メソッド呼び出しやフィールドアクセスなどがジョイ

ンポイントになる．ポイントカットとは，すべてのジョインポイントの中からある特定の条件を満たすポイントを選び出す機能である．AspectJでポイントカットをcall（* *.set*(..))とすると，名前がsetで始まるメソッドの呼び出しポイントを選び出してくれる．アドバイスとは，ポイントカットによって抜き出したジョインポイントでのプログラム実行を変更する機能である．ジョインポイントの前後（before/after）にコードを挿入したり，実行すべきコードを置き換え（around）たりできる．たとえば，ポイントカット指定call（* *.set*(..)）に対してメッセージ表示のアドバイスを指定すれば，名前がsetで始まるメソッドが呼び出されるたびにメッセージが表示されるようになる．AspectJでは，アスペクトはポイントカットとアドバイスを記述したモジュールとして定義される．JPMに基づいた言語処理系のことをアスペクト指向ではウィーバ（weaver）と呼ぶ．

アスペクト指向を適用すると，組み込みソフトウェアにおいて重要なチェック処理や同期処理などをアスペクトとしてモジュール化することが可能になる．以下に，AspectJによる記述例を示す．このアスペクトは，名前がmodeで始まるメソッドの呼び出しの前で必ずモードが妥当かどうかをチェックするためのものである．このアスペクトをディジタル時計のモデルに適用すると，mode_button操作が実行されるたびに妥当性のチェックが行われる．

```
public aspect CheckMode {
 pointcut modeOperation():call(* *.mode*(..));
 before(): modeOperation() {
  // モードのチェック
 }
}
```

MDAへの応用

AspectJなどのプログラミング言語は非機能要求の記述に有効であるが，そのままではMDAに適用できない．MDAに適用するにはコードレベルではなくモデルレベルでアスペクトが記述できる必要がある．それには2つの問題を解決しなければならない．1つは，アスペクトのダイアグラム表記に関する問題である．もう1つの問題は，ウィーバ機能を持ったモデルコンパイラの開発である．最初の問題については，UML2でもまだアスペクト図はサポートされていないが，UMLにアスペクトを導入する試みはすでになされている．また，基本的に表記に関する問題に過ぎず，標準化への合意が取れれば技術的には大きな問題はないと考えられる．それに対し，もう1つの問題は解決しなければならない点が多いが，すでにいくつかの研究がなされている．その1つが，Grayらが開発しているAODM（Aspect-Oriented Domain Modeling）である[14]．AODMではアスペクトを記述するのにECL（Embedded Constraint Language）という言語を導入している．ECLはOCLを拡張するとともにQVTのアイディアを取り入れた言語である．属性や関連などのモデル要素を追加するなどの機能を持つ．QVTでは横断的関心事に対する変換機能はないが，ECLではそれが可能になっている．

以下は，ECLによるアスペクトの記述例である．モデルからメソッド名がcomp で始まるメソッド群をポイントカットにより抽出し，それにプロセッサを割り当てるための定義である．これによって，元のモデルに変形が加えられる．ディジタル時計のモデルにこのアスペクトを適用すると，メソッドcompWorldTime の実行に新たなプロセッサを割り当てることができる．

```
pointcut ProcessorAssignment {
  models("")->select(m|m.kind()= "comp*")
            ->Assign();
}
```

アスペクト指向による新たなMDAの実現

アスペクトには2つの側面が存在する．1つは，ソフトウェアモジュールとしての側面である．システムは機能を表現したクラスモジュールと横断的関心事を表現したアスペクトモジュールから構成される．もう1つは，変換モジュールとしての側面である．アスペクトの記述をみると，「ポイントカットで指定した個所にアドバイスにより変換を行え」といった操作の記述と捉えることができる．この二面性は非常に重要である．通常のMDAでは，モデルを作成する人とモデル変換規則を作成する人は別々であるが，アスペクト指向の考えを導入することにより，これらを同じ人が，同じモジュール化メカニズムを用いて作成することが可能になる．また，変換モジュールとしてのアスペクト部品を整備することにより，対象となる組み込みソフトウェアの特性に応じたモデルコンパイラの構築も可能と考えられる．

形式手法による組み込みソフトウェアの検証

組み込みソフトウェアの正しさをテストだけで保証することは困難である．従来，このような目的に形式手法が用いられてきた．システムが正しく動作することを検証するために用いられる形式手法の1つとして，モデ

ル検査[15)]がある☆1．モデル検査は有限の状態空間に対する網羅的探索によって，システムがある性質を満たすことを自動的に検査する手法である．調べたい性質はCTL（Computation Tree Logic）やLTL（Linear Temporal Logic）などの時相論理式（temporal logic formula）で記述される．モデル検査の手法をモデリング段階に適用しようという研究がある．Flakeらは，OCLを拡張したCCTL（Clocked CTL）を提案している[16)]．CCTLはCTLに時間境界（time-bounded）を追加したもので，[a,b]（min a max b: aからbの間）といったことが指定できる．

ディジタル時計の例で考えてみよう．「どのモードからでもいつかは分設定中（Set minutes）の状態に遷移でき，分（minutes）の値を30に設定できる」という性質は以下の時相論理式で表現できる．A（ll），G（lobal），F（uture），は論理演算子である．この論理式は，いつかは（F演算子），分の値が30に設定される状態に（minutes＝30），すべてのパス上の（A演算子），すべての状態から（G演算子）から到達可能であることを示す．

```
AGF(minutes=30)
```

モデル検査を適用する際に問題となるのは状態数の爆発をどう抑えるかであるが，プログラミングコードレベルでモデル検査を適用するよりモデリング段階で適用した方がよい場合がある．コードレベルでは消失してしまうような設計情報を用いることによりモデリング段階での検証をさらに効率化することが考えられる．実装に対しモデルは抽象であり，このレベルでモデル検査を行い正当性が保証されれば，モデル変換に誤りがない限り，生成されたプログラムは正しい．生成されたプログラムよりもモデルの方が状態数が少ないので，効率的な検証が可能になる．これは，一種の抽象モデル検査と考えることができる．抽象モデル検査とは状態遷移システムを抽象写像によって抽象システムに写し，抽象システムを従来の方法でモデル検査することによって，元の状態遷移システムを検証する方法である．

まとめ

本稿では，組み込みソフトウェア開発手法の変遷を辿るとともに，次世代の開発手法として，組み込みソフトウェアをモデル駆動で開発する方法をアスペクト指向や形式検証などの技術と絡めながら紹介した．現在のMDAではPIMからPSMの部分が中心となっているが，もう少し範囲を広げて開発者の行為を観察すると，「モデルの作成とモデル間の変換」が連続した作業として抽象化できる．そうすると，個々のモデルの表記法を統一し，モデル間の変換規則を記述するための言語を統一すれば，開発過程を自動化できるのではないかと考えるのは自然である．モデル駆動開発の発想もここにあり，MDAはその部分解といえる．実は，モデル駆動開発の考え方を一般化した研究が1980年代からすでに存在する．Neighborsによって提案されたDraco[17)]である．Dracoでは変換そのものを部品にしようという考え方をとっていた．本稿で述べてきたように，モデル駆動開発を実現するには，アスペクト指向やプログラム変換，モデル検査，などといったプログラミング言語の研究で培われてきた技術が鍵となる．UMLも言語として成長しつつあり，今後この方向性はますます強まるのではないかと思われる．

参考文献

1）CMU/SEI : Product Line Approach to Software Development, http://www.sei.cmu.edu/plp/
2）千葉　滋：アスペクト指向ソフトウェア開発とそのツール，情報処理 Vol.45, No.1, pp.28-33（Jan. 2004）.
3）UML, http://www.omg.org/uml/
4）渡辺政彦，飯田周作，石田哲史，山本修二，浅利康二：UML動的モデルによる組み込み開発，オーム社（2003）.
5）Mellor, S. and Balcer, M.: Executable Uml: A Foundation for Model-Driven Architecture, Addison Wesley（2002）（翻訳：Executable UML MDAモデル駆動型アーキテクチャの基礎，翔泳社（2003））.
6）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）.
7）Eriksson, H., Penker, M., Lyons, B. and Fado, D.: UML 2 Toolkit, OMG Press（2004）.
8）MDA, http://www.omg.org/mda/
9）Warmer, J. and Kleppe, A. : The Object Constraint Language Second Edition − Getting Your Models Ready for MDA, Addison Wesley（2003）.
10）QVT, http://qvtp.org/
11）Sztipanovits, J. and Karsai, G.: Generative Programming for Embedded Systems, Proceedings of International Conference on Generative Programming and Component Engineering（GPCE 2002），pp.32-49（2002）.
12）AspectJ , http://www.eclipse.org/aspectj/
13）Kiczales, G.: The Fun Has Just Begun, Keynote Talk at International Conference on Aspect-Oriented Software Development（AOSD 2003），（2003）.
14）Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale, A. and Natarajan, B.: An Approach for Supporting Aspect-Oriented Domain Modeling, Proceedings of International Conference on Generative Programming and Component Engineering（GPCE 2003），pp.151-168（2003）.
15）Clarke, E., Grumberg, O. and Peled, D.: Model Checking, The MIT Press（1999）.
16）Flake, S. and Mueller, W.: An OCL Extension for Real-Time Constraints, T. Clark and J. Warmer（eds.）: Object Modeling with the OCL, Vol.2263 of Lecture Notes in Computer Science, Springer-Verlag, pp.150-171（2002）.
17）Neighbors, J.: The Draco Approach to Construction Software from Reusable Components, IEEE Transactions on Software Engineering, Vol. SE-10, No.5, pp.564-573（1984）.

（平成16年6月4日受付）

☆1 モデル検査の詳細については，本特集の「組み込みソフトウェアのモデル検査技術入門」を参照されたい．