修 士 論 文

組込みシステム向けMDA開発環境の研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

細合 晋太郎

2007年3月

修 士 論 文

組込みシステム向けMDA開発環境の研究

指導教官　岸知二 特任教授

審査委員主査　岸知二 特任教授
審査委員　片山卓也 教授
審査委員　落水浩一朗 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

510090 細合 晋太郎

提出年月: 2007 年 2 月

概 要

近年組込みシステムの需要が大幅に増えてきている. しかしながら現在の組込みシステム開発は, 高度な要求に対し大規模化したソフトウェアを複雑に変化するハードウェアに合わせて作成し, なおかつ高信頼性, リアルタイム性を保持しながら非常に短期間で開発しなくてはならない, という非常に厳しい状態にある. 従来の開発手法では, このような現状に対応することは難しく, 新たな開発手法が求められている.

このような問題に対し新たな開発手法として,MDA(Model Driven Architecture) に注目が集まっている. MDA は OMG の提唱する UML 等のモデルを開発の流れの中心とする開発手法であり, PIM/PSM によるモデルの再利用性の向上や, モデル変換, コード生成により開発を効率化することができる.

現在, すでに組込みシステム開発向けにもいくつかの MDA ツールが導入され始めてきている. これらのツールは非常に有効であるが, 頻繁にハードウェア面からの変更が伴い, かつハードウェアリソースの最適化が必要な情報家電などの小〜中規模システムにおいては, ツールベンダが対応していないデバイスが使えない, ハードウェアとの連携が取りにくいといった問題がある. またカスタマイズを行うといった際にも, 従来の MDA では小回りが利きにくい.

本研究は従来の MDA に加え, ハードウェアの情報もモデルとして取り入れることで, ハードウェアとの協調が欠かせない組込みシステム開発に即した MDA を提案するものである.

本論文では, まず必要となるハードウェア情報の整理を行い, 効率的にモデル化が行えるようメタモデルを定義した. またハードウェアの変更に耐えうる開発の流れとするため, HW PIM, HW PSM, SW PIM, SW PSM, LIM といったモデルの定義と各モデル変換の定義を行った.

次に Eclipse 上に Plugin として提案手法の実装を行った. さらに, 提案手法の評価を行う為に, デジタル時計の例を用いて作成したツール上で開発を行った.

ハードウェア情報を取り込むことで, 従来手動で記述しなくてはならなかった部分のコードの自動生成を確認することができた. またモデル化することにより, 散在していた情報を一元的に取り扱え, 情報自体の資産化や共有にも有効であることが確認できた.

目 次

図 目 次

第1章　はじめに

1.1　背景

近年, 組込みシステムの需要が高まってきている. 組込みシステムの開発では, のハードウェア, ソフトウェアと両面の技術が必要となってくる.

携帯電話などの大規模システムでは, 年々増えるソフトウェア規模が問題となっているが, プラットフォームはある程度固まっており Linux や iTron といった OS が利用出来るため, 比較的汎用コンピュータに近い開発を行うことができる.

しかしながら情報家電などの中小規模のシステムでは, 品種や製品ごとにハードウェア構成が異なり, 構成が変わるたびにそれに合わせたソフトウェアの開発が必要となる. また開発の期間も非常に短く, 従来の開発手法では対応することが難しくなってきている.

ハードウェアを操作するようなソフトウェアの開発を行う場合, 対象となるハードウェアの情報を参照しながら進めていく必要がある. ハードウェア開発においては, これらの情報は電子化され開発の流れの中に取り込まれているが, ソフトウェア開発では図表といった人が読み理解しなくてはならない情報のまま散在している.

現在組込みシステムの開発を効率化するため, MDA(Model Driven Architecture) という開発技法に注目が集まっている. オブジェクト指向をさらに進め, モデルを中心に開発を行い, 一つのモデルから様々なプラットフォームに合わせたモデルの生成, モデルからのコード生成といったことが可能となる.

しかしながら, 現在の MDA では対象とするプラットフォームの抽象度は, OS やミドルウェアといったソフトウェアレベルのもので, ハードウェア構成の変化に対応することは難しい.

1.2　目的

本研究の目的として, 中小規模の組込みシステムを対象とし,

- ハードウェア構成の変更に耐えうる開発手法の提案
- 開発工程に散在する情報のモデル化による利用性の向上
- コード生成による開発速度の向上

が挙げられる.

1.3 アプローチ

本研究のアプローチとして, まずソフトウェア開発から必要となるハードウェア情報を体系化し, それらをモデルとして扱う為のメタモデルを作成した. 次に組込みシステムに即した MDA 開発の流れとなるよう, モデルの定義とモデル間の変換の定義を行った.

1.4 本論文の構成

2 章で対象となる組込みシステムの概要と現状について述べ, 3 章では中核となる MDA の概要並びに関連技術について述べる. 4 章は前 2 章での問題点を元に, 改善点を整理し提案手法について詳細に述べる. 5 章では提案手法を元に行った実装を行ったシステムに関して, その構成や用いた技術について述べる. 6 章は作成したツールを用いて簡易な例への適用並びにその評価について述べる. 7 章で本研究の考察と今後の課題についてまとめ本論文を総括する.

第2章　組込みシステム

本章では、研究の対象とする組込みシステムについて述べる

2.1　組込みシステムとは

組込みシステムとは、機器の制御にコンピュータを用いるシステムで、コンピュータが機器に組込まれていることから組込みシステムと呼ばれる。一般に汎用のコンピュータ機器以外のコンピュータを内蔵しているシステム全般を指す.

組込みシステムの例として,

- 情報家電 (テレビ, ビデオ, オーディオ)
- 白物家電 (エアコン, 冷蔵庫, 洗濯機, 電子レンジ, 炊飯器, ポット)
- 携帯端末 (携帯電話,PDA, 電子辞書)
- 自動車内部 (ECU,ABS, パワーステアリング, パワーウィンドウ, エアコン)
- 自動車周辺 (カーナビゲーションシステム, カーオーディオ,ETC)
- 設備機器 (エレベータ, 空調システム, 自動ドア, 自動販売機)
- 工業機器 (プラント制御, 工作機械, 工業用ロボット)

など, 例として挙げるだけでも非常に多岐に渡るシステムが含まれる.

システムの規模に関しても,4bit といった低機能の MCU(Micro Control Unit) を用いたポットなどの家電から, 32bit,64bit といった汎用コンピュータ並みの高機能 MCU を搭載した携帯電話やカーナビゲーションシステムなど様々である.

主な組込みシステムの特徴として,

- リソースが乏しい
- リアルタイム性が必要
- 高信頼・高品質が求められる
- ハードウェアプラットフォームが変化し易い

- ハードウェアとの協調が必要

などが挙げられる.

2.2 組込みシステム開発の難しさ

組込みソフトウェアは汎用のコンピュータ上のソフトウェアとは違い, ハードウェアを直接制御する必要があり, より高い信頼性やリアルタイム性が求められる. 高度化する要求に伴い, ソフトウェア規模は年々大規模化してきているが, 開発手法は依然として旧来のものを使い続けている現場が多い.

組込みシステムのハードウェア構成は, 直接製品コストに繋がる為, 要求を満たす最小限のハードウェア構成が求められる. ハードウェア・ソフトウェアの両方を同時に作り始めることも多く, ソフトウェア規模や, 必要なハードウェアリソースの予測が必要となる.

また, ハードウェアを高度に制御する為, 非常に低レベルの部位のソフトウェアを作る知識も必要となってくる. さらに新製品開発のスパンも短く, 非常に短い期間でハードウェア, ソフトウェアの両方を開発していかなくてはならない.

2.3 対象とする組込みシステム

本研究では, 情報家電や車載システムなどのハードウェア構成の組み合わせが多く, 短期間の開発が望まれる中小規模の組込みシステムを対象とする.

第3章　Model Driven Architecture

本章では, 提案の中核をなす MDA について述べる.

3.1　概要

MDA とは OMG の提唱する, モデルを中心とした開発技術である.

従来の開発サイクルでは, モデルは図式として用いられ, それを人間が読み理解しプログラミング言語に翻訳していた. MDA では,UML 等の設計に用いられてきたモデルをプログラミング言語のように扱う.

MDA では, 機械語からアセンブラ、アセンブラから C 等の高級言語, 高級言語から Java などのオブジェクト指向言語と移ってきたように, オブジェクト指向言語からモデリング言語を入力とした開発に移ろうとしている. 人間に理解し易い図式等を入力とすることで, 抽象度を高く保ったまま, 大規模な開発が行える.

MDA ではメタモデル (モデルを定義するためのモデル) というものを用いることで, モデル自体の定義を行う. さらにメタモデルはメタメタモデル (MOF) の要素を用いて定義されている. このようなメタな要素を取り入れることで, モデル間の変換やモデル自体の情報を体系だって取り扱うことができる.

また PIM(Platform Independent Model:プラットフォーム独立モデル), PSM(Platform Specific Model:プラットフォーム依存モデル) を切り分けることにより一つの PIM から複数の PSM を自動生成することが出来る. これにより, 複数の環境に適用可能なソフトウェアを構築することが出来る. また将来の仕様変更に対しても新たに PSM を生成することでき, ライフサイクルの長いソフトウェアモデルとなる.

3.2　モデル

MDA の入力となるモデルは, 一般に UML が用いられることが多い. これは UML がソフトウェアを記述するのに適していること, UML 自体が MOF を用いて定義されていることなど,MDA に適したモデルであるからである.

しかしながら, 必ずしも UML を用いる必要はなく, MOF の要素を用いて定義したメタモデルを持つモデルであれば, MDA の入力モデルとして扱うことが可能である.

本研究では,UML だけではハードウェア情報を記述するのは難しいと思われたため, 新たなモデルとして DDM,HW PIM,HW PSM といったモデルを導入した. またこれらのモデルを定義する為のメタモデルとして DDMM の定義を行った. これらの詳細については 4 章で述べる.

3.3 メタモデル

MDA では, モデルを定義する為のメタモデルという概念を取り入れている.

これは,XML 文書に対する XMLSchema のような位置付けであり, モデルを構成する要素の定義を行う上位概念である.

UML においても, メタモデルで構成要素が定義されており [4], 例えばクラス図でのクラスの要素は, メタモデルにおいて Classifier(識別子を持つ) を親クラスとした,Class クラスとして定義されており, 包括する要素として Property(フィールド) や,Operation(メソッド) といったものも定義されている.

さらに, メタモデルの要素はメタメタモデル (MOF:Meta Object Facility) で定義されていが, MOF は MOF を用いて定義されているので,MDA ではこれ以上の上位概念はない.

これらのメタモデルの階層構造として,MDA では 4 層メタモデルを定義している.

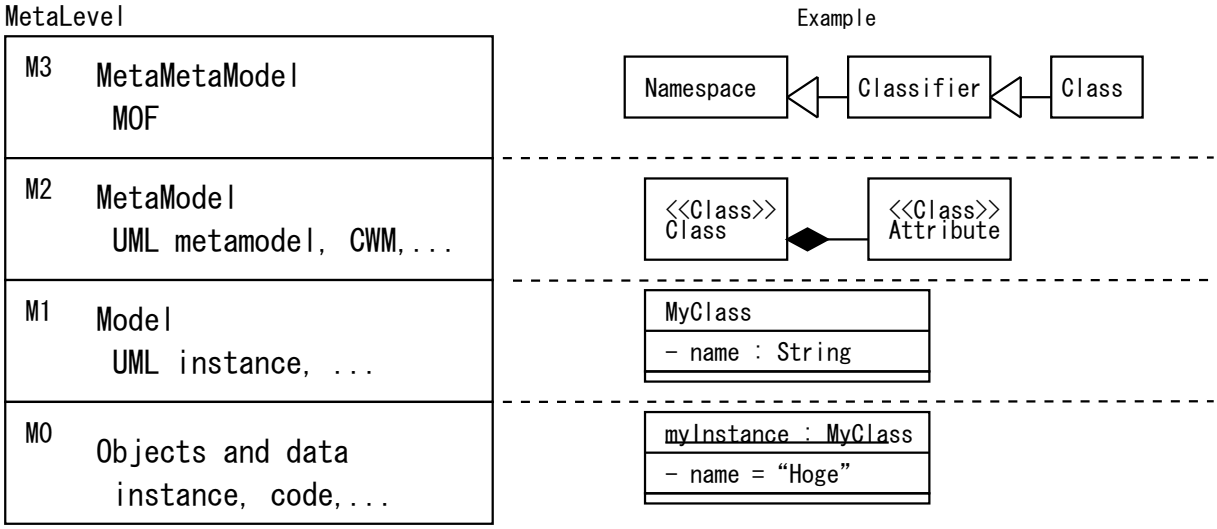

図 3.1: 4 層メタモデル

MDA でモデル変換やコード生成といったことを行う場合, モデル自体の構造や情報が不可欠となる. このようにメタ要素をアーキテクチャとして取り入れることで, 上位の概念を把握するだけで, 下位のモデルを容易に扱うことができる.

3.4 PIM/PSM

MDA の重要な概念の一つとして, PIM/PSM が挙げられる.

モデルを特定のプラットフォームから独立した形で定義しておくことで, 可搬性が高くライフサイクルの長いモデルとすることができる.

PIM から PSM への変換には, 変換の為のルールが必要となるが, 一度そのプラットフォームへの変換ルールを定義してしまえば, PIM に加わったり, 新規に PIM を作成した場合でも PIM から PSM を自動生成することが可能となる.

また複数のプラットフォームへの移植性を高めるだけでなく, プラットフォームのアップデート等に対しても有効であり, 従来のソフトウェアに比べ非常にライフサイクルの長いものとなる.

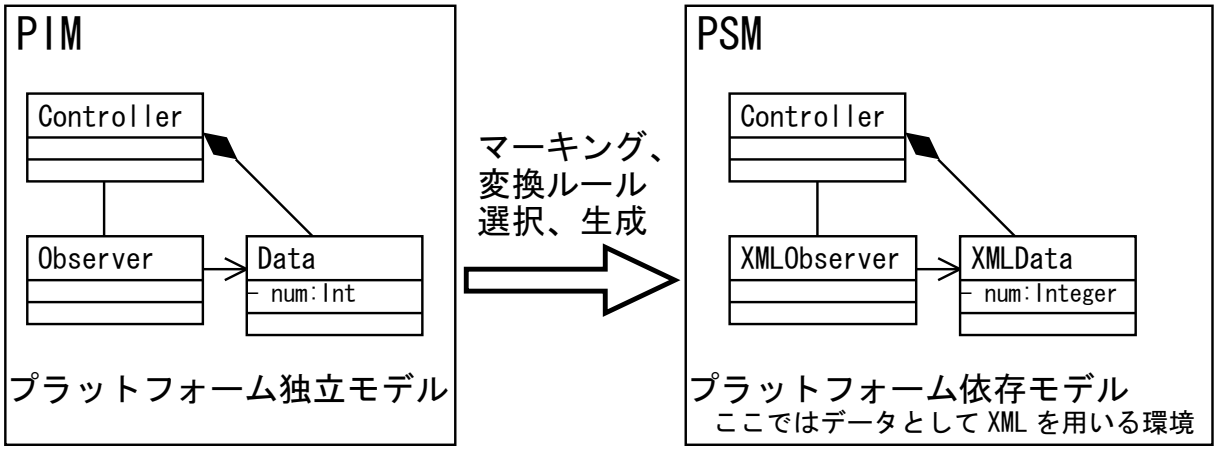

図 3.2: PIM/PSM

3.5 モデル変換

MDA を実際に行う上で必要となってくるのが, モデル間の変換である.

PIM から PSM への変換や,UML から CWM への変換といったモデルからモデルへの変換に加え,UML からコードへの変換もモデル変換の一種と考えられる.

モデル変換のルールは, メタモデルのレベルで行い, どの要素が別のモデルのどの要素にどのように変換するのか, といった情報を定義していく.

また変換を行う際には, どの要素をどのモデルに変換するのかといったマーキングの情報も必要となってくる.

本研究では, モデル変換を行うために,oAW という Eclipse Plugin の MDA フレームワークを用いている.

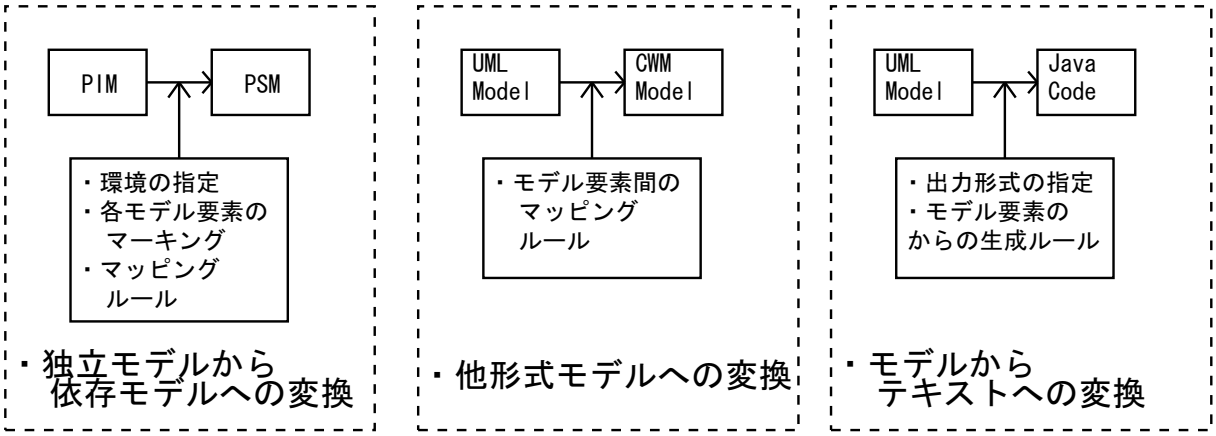

図 3.3: モデル変換

3.6 既存技術

すでにいくつかの組込み向け MDA ツールが提供されている.

3.6.1 Bridge Point

MentorGraphics 社製の MDA ツール. 方法論は executableUML に沿う. システムの構造などはクラス図等の図を用いて定義し, 振舞いに関する部分は ActionSemantics という特定の環境に依存しない言語を用いて定義する.

記述した UML はツール上で実行することが可能で, コードに変換する前にモデル上で動作の確認を行うことができる.

また生成をサポートする言語として,C/C++などがある.

3.6.2 Rational Rose Technical Developer

IBM 社製 MDA ツール. 方法論は ROOM に沿う.

シーケンス図, ステートマシン図などからのコード生成をサポートする. また,VM 上での UML でのシミュレーションも行える.

生成をサポートする言語として,C/C++,Java などがある.

3.6.3 Rhapsody

Telelogic 社製の MDA ツール. 方法論はリアルタイム UML に沿う.

UML2.0 のすべての図をサポートし, クラス図とステートチャート図, シーケンス図などからのコード生成やシミュレーションをサポートしている.

また, 生成できる言語として Java, C, C++, ada などがある. リバースエンジニアリングにも対応している.

3.7 既存技術の問題点

上記のツールのいずれも比較的大規模なシステムを対象としており, ハードウェア構成の変化が大きい中小規模で扱うことは難しい.

中小規模のシステムで用いる場合の問題点として以下のものが挙げられる.

ツールベンダの対応するデバイスしか扱えない 多くのツールではデバイスに対応するコード生成は, ミドルウェア等を利用したものとなっている. このため対応していないデバイスを用いる為には, その部分のコードをユーザが実装しなくてはならない.

ハードウェアに適したカスタマイズを行うことが難しい. 組込みシステムでは, 最小限のハードウェアで最大限のパフォーマンスを求めるような, 非常に高度なカスタマイズが要求される. 生成されるコードは高機能なものではあるが, 必ずしもそのシステムでの構成に最適なものとは限らない.

ハードウェア構成の変化に対応できない. 従来の MDA で吸収できるプラットフォームの差異は, ほとんどが OS やミドルウェアレベルのものである. ハードウェア構成自体が変更となった場合, それに対応するためにはユーザ側で複雑なルールを定義するか, コードの実装を行うことになる.

ハードウェア開発との連携が取りにくい ほとんどの MDA が主にソフトウェア開発を対象としたものであり, ハードウェア開発との連携は意識されていない.

第4章　提案

本章では, ハードウェア情報をモデル化し,MDA に取り入れる提案手法について述べる.

4.1　ハードウェア情報のモデル化

組込みシステムの開発において, ハードウェアとの接点となる部分のソフトウェアコードは, 回路図などの情報を元に,MCU のどのピンにはどのデバイスが繋がっているといったことを確認しながらコーディングを行っていく.

MCU の I/O ポートを用いる為には, そのポートをどのように用いるか, 入出力方向はどちらであるか, といった初期化の処理も必要となってくる.

しかしながらこれらのコードは,MCU ごとにある程度決まっており, 一定のパターンに沿うものである. またハードウェアデバイスの操作についても, 使用法は仕様書にて定義されている.

従来はこれらは人が読み, 人の手でコーディングを行わなければならなかった. 当然ハードウェアに直結した部分のコードであるだけに, 不具合が含まれると, ハードウェアを破壊する危険性も伴う.

このようなハードウェア情報をうまく参照することができれば, 自動化を行う余地がある部分に対して, 本手法ではハードウェアのモデル化を行い,MDA に取り込むことでこれらのコードの自動生成を試みるものである.

ハードウェア情報のモデル化は, コードの自動生成に限らず, クロックやメモリ量といったハードウェアのパフォーマンスに関わる情報や MCU に内包されている機能の有効利用等, 様々な側面で利用価値があると思われる.

今回は特に, 組込みシステム開発では毎回のように必要になるであろう,MCU の初期化の部分と, デバイスドライバの生成に注目し, 必要となるハードウェア構成とハードウェアデバイス自体の情報をモデル化する.

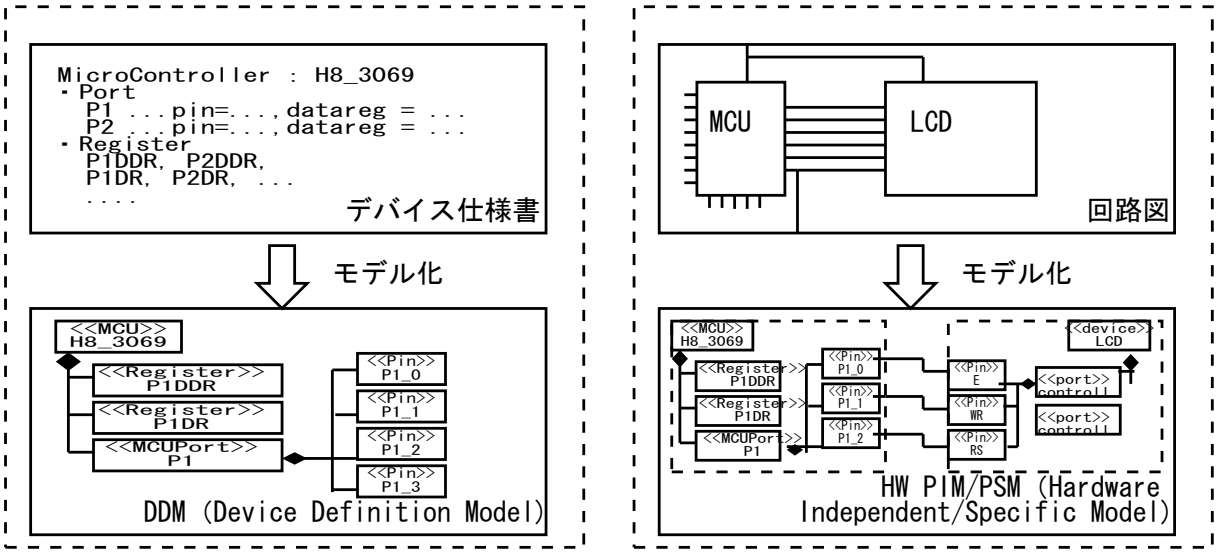

図 4.1: 回路図, 仕様書のモデル化

4.2 全体像

個々のモデル・メタモデルの解説に入る前に, 提案全体のモデルの流れを以下に示す.

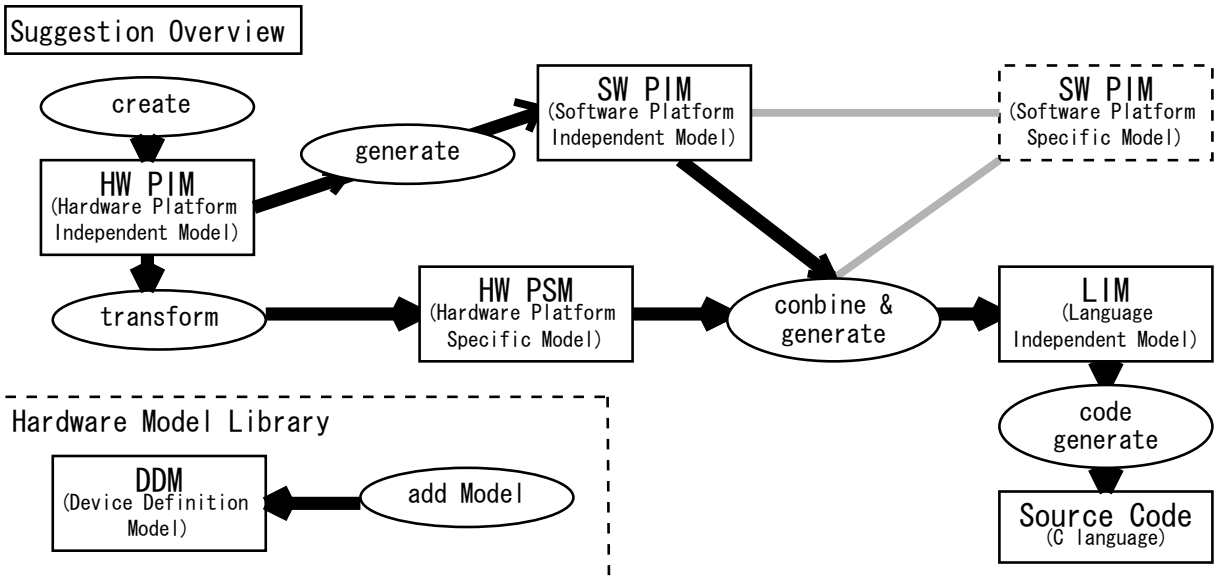

図 4.2: モデルの位置付けと流れ

本提案の流れとして, まず, 左下部の DDM でデバイス自体の情報をモデル化する. このモデルはデータベース的な役割を果たし, 主にハードウェア担当者がモデルの定義を行い, 追加する. ハードウェアの振舞いに関しては, ソフトウェア設計者がシステムに合わせて抽象化したインターフェイスを元に, 必要となる振舞いを定義する.

次に左上部の HWPIM において, 抽象的なハードウェア構成を定義する. この部分の定義は, システム全体を設計する担当者 (以下システム設計者) が行う.

HW PIM からは,HW PSM(詳細なハードウェア構成) と SW PIM(ソフトウェアモデル) のスタブモデルの生成を行うことができる.

SW PIM はソフトウェア担当者に渡され, 詳細な設計を行っていき, HW PSM はシステム設計者とテスト担当者が必要となるリソースに合わせ調整していく.

SW PIM, HW PSM からは,LIM(コード独立の完全モデル) の生成を行うことができる. 基本的に LIM に変更を加える必要はなく, 各言語やシミュレーション環境へのコード生成に用いられる.

4.2.1 典型的な分担例

以下に各モデルを扱う担当者を示す.

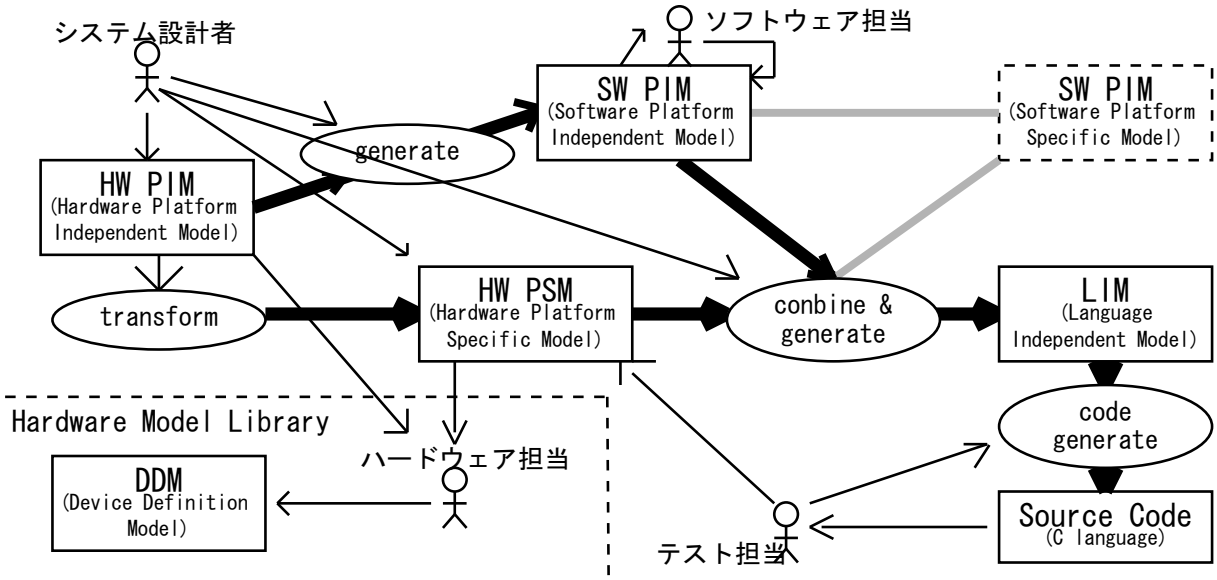

図 4.3: 典型的な分担例

4.3 本研究で用いるモデル

本項では各モデルの説明とモデルを定義するためのメタモデル説明を行う.

4.3.1 DDMM(Device Definision Metamodel)

DDMM(デバイス定義メタモデル) は後述する DDM,HW PIM, HW PSM に対するメタモデルである。

DDMM は以下の 3 つのメタモデルから構成されている.

- Structure
  DDMM Structure では, ハードウェアデバイスそのものの情報と, システムにおけるハードウェア構成の情報を扱う為のメタモデルを定義している. メタモデル内部では, ハードウェアデバイスを MCU(制御側要素),Device(被制御要素) の 2 つに大別している. これはシステムをソフトウェアの面から見た際に, ソフトウェアを内包する要素とソフトウェア外部に位置する要素を明確に分ける為である.

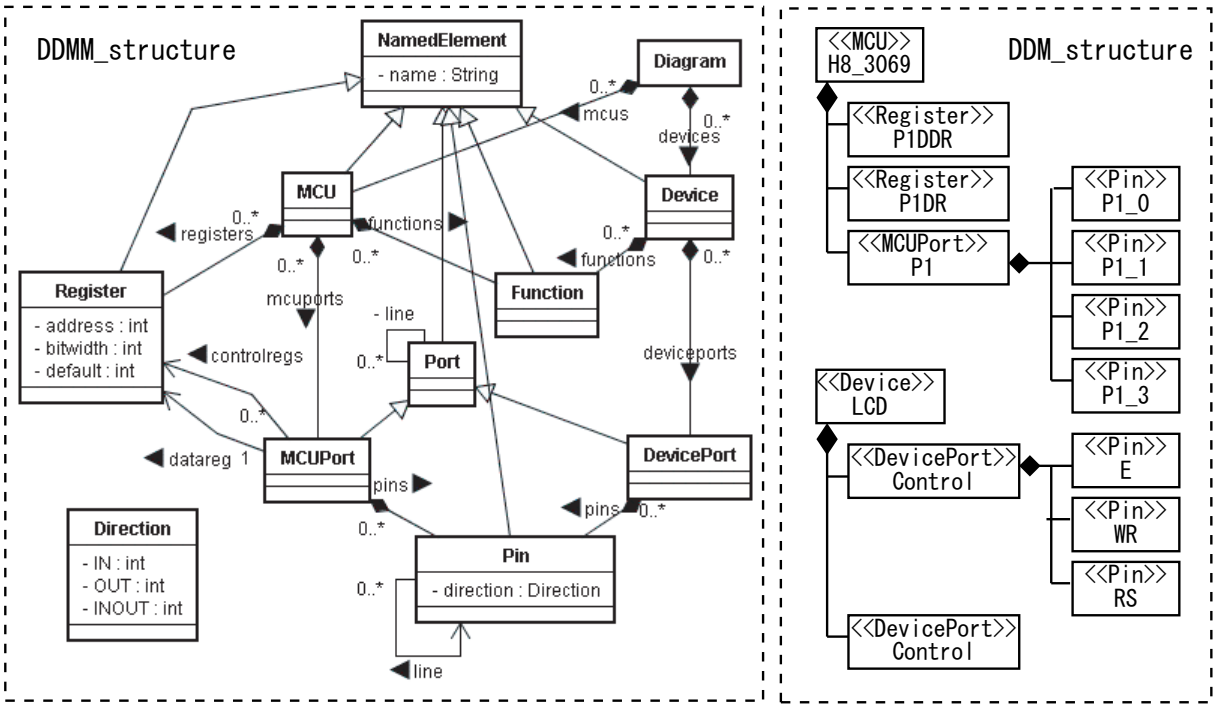

図 4.4: DDMM Structure メタモデルとモデル例

- Behavior
  DDMM Behavior では, ハードウェアデバイスの操作方法に関する情報をモデル化するためのメタモデルを定義している.

メタモデルの構成要素はUMLのアクティビティ図要素のサブセットとなっている. ハードウェアデバイスは, 初期化のシーケンスや通信のためのプロトコルなど予め決まった手順に従い操作を行うこととなる. これらの操作方法をモデル化することにより, これらの操作法に対するソフトウェアコードの生成を行う.

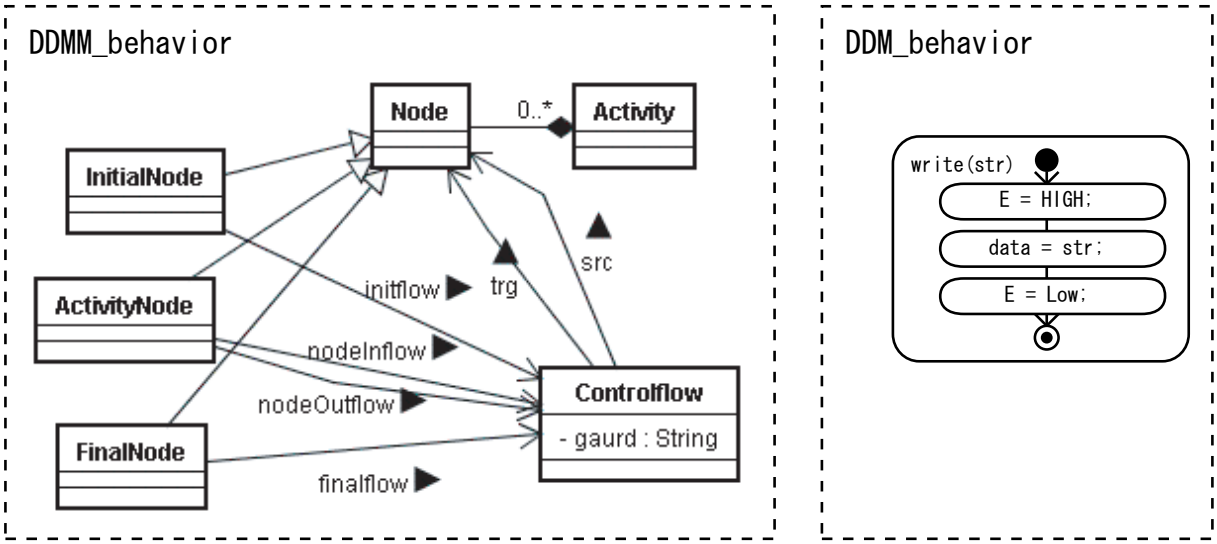

図 4.5: DDMM Behavior メタモデルとモデル例

- Category
  DDMM Category では, ハードウェア情報を階層的にまとめる為の情報を定義するためのメタモデルを定義している.

  デバイスやMCUは, 種別やメーカ, シリーズなどによってある程度の共通性を持っている. これらを階層的に取り扱うことにより, その階層に属する新規デバイスの追加などの際に重複する情報を効率的に取り扱うことが出来る.

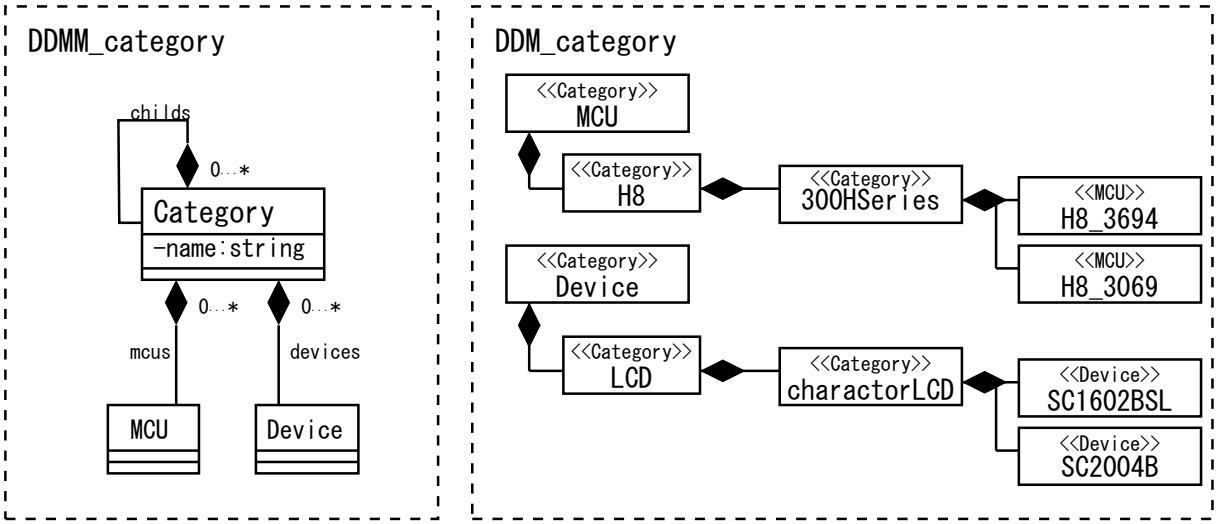

図 4.6: DDMM Category メタモデルとモデル例

4.3.2 DDM(Device Definition Model)

DDM(ハードウェア定義モデル) では, ハードウェアデバイスそのものの情報を取り扱う. DDM は DDMM の全メタモデルに従う.

具体的には, MCU であれば I/O に関するポートの情報, レジスタの情報,MCU 固有機能などをモデル化している.

また, 各 Device の機能ごとに,DDMM Behavior の要素の Activity を持ち, シグネチャに対する振舞いを定義している.

さらに DDMM Category の要素を用いて複数の DDM をまとめたデータベースを構築している. また,Category の要素に対応する Structure の要素は, 抽象デバイスとなる.

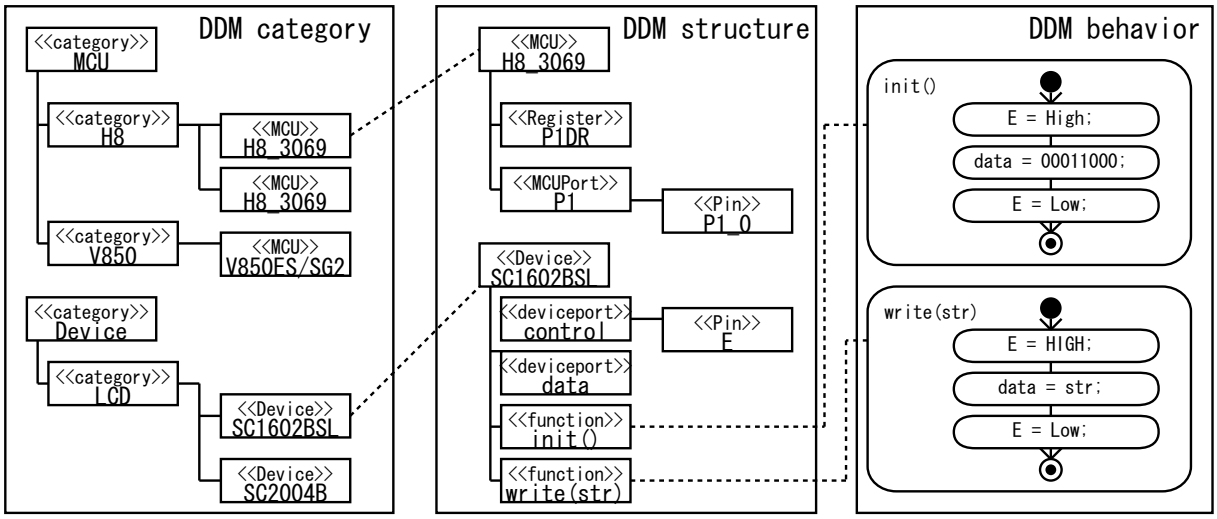

図 4.7: DDM モデル例と各モデル間の関連

category で, すべてのデバイス・抽象デバイスをツリー構造として表す. 各デバイスは structure と関連しており, 個々に詳細な構造が定義される. また,structure で定義されたメソッド (シグネチャ) に関連する形で, behavior でその実装が定義される.

4.3.3 HW PIM(Hardware Platform Independent Model)

HW PIM(ハードウェア プラットフォーム独立モデル) では, 特定のハードウェアデバイスや MCU に依存しないハードウェア構成モデルを扱う.

要素として,DDM category の Category 要素, DDM structure の MCU,Device の抽象モデルを用いる.

抽象化されたデバイスでハードウェア構成を記述することで, ハードウェア変更に対して独立したモデルとなる.

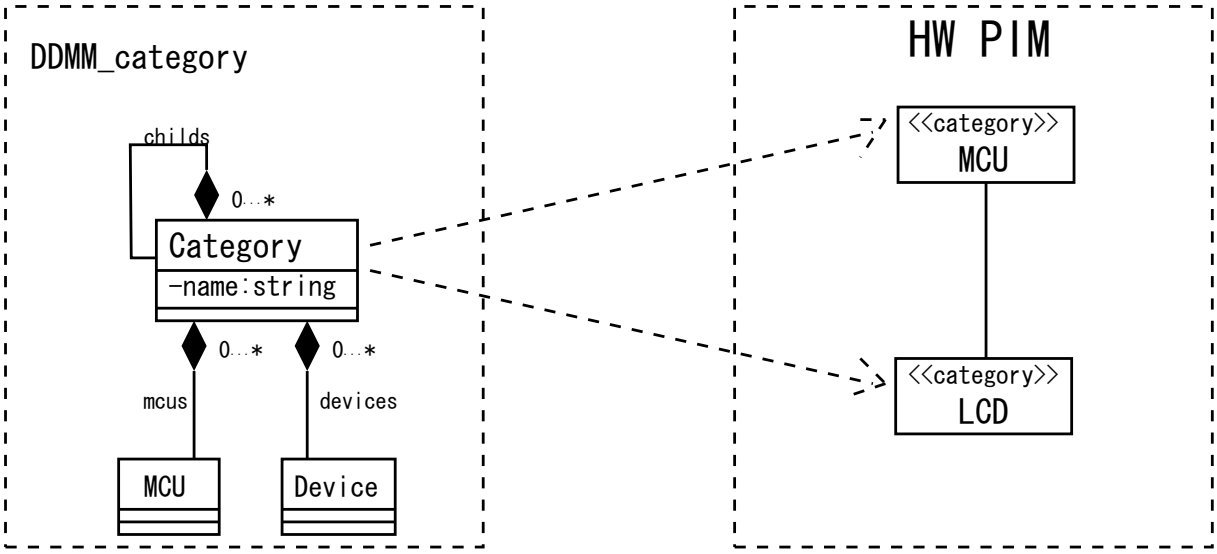

図 4.8: HW PIM メタモデルとモデル例

4.3.4 HW PSM(Hardware Platform Specific Model)

HW PSM(ハードウェア プラットフォーム依存モデル) では, HW PIM の要素を特定のデバイス,MCU に結びつける.

またデバイス・MCU が決定されることで, ポート情報などの詳細なハードウェア情報も決定され, それらの情報を元により詳細なハードウェア構成モデルを記述していく.

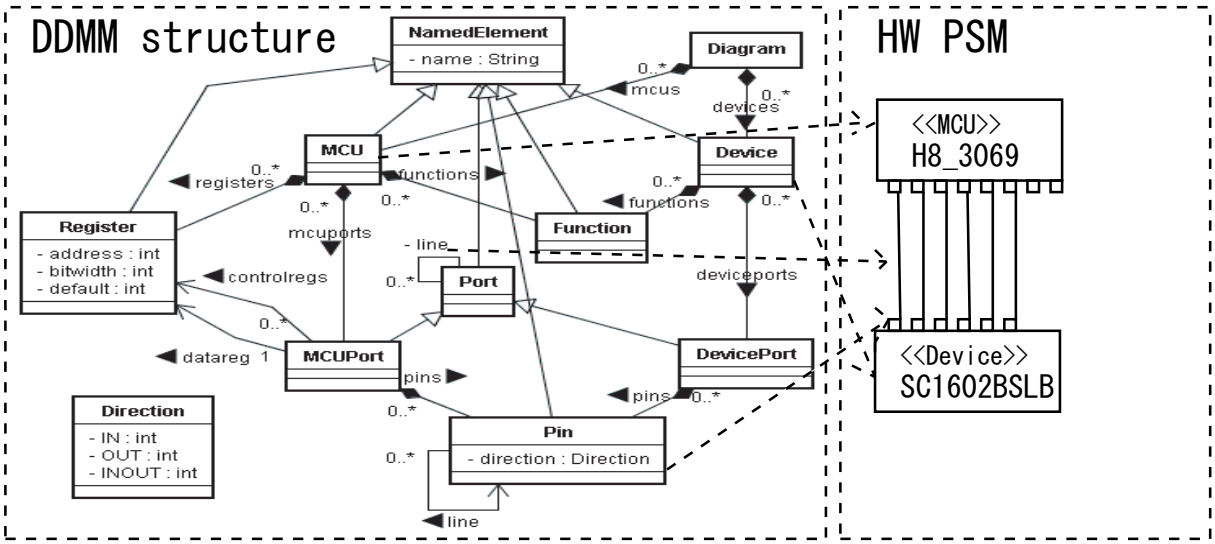

図 4.9: HW PSM メタモデルとモデル例

4.3.5 SW PIM(Software Platform Independent Model)

SW PIM(ソフトウェア プラットフォーム独立モデル) では, 特定の OS などに依存しないソフトウェアモデルを記述する.

ハードウェアとの接点となるデバイスドライバの部分は, HW PIM からインターフェイスとして生成される.

定義は行っているが今回の実装では,取り扱う範疇が広くなり過ぎるため実装を見送った.

モデルは UML の要素である.

4.3.6 SW PSM(Software Platform Specific Model)

SW PSM(ソフトウェア プラットフォーム依存モデル) では, 特定のソフトウェア環境 (OS 等) に依存したモデルを記述する.

SW PIM 同様, 今回はコード生成の対象とはせず, HW PIM からのスタブモデルの生成を行うに留まった. また,SW PIM, SW PSM は同一のモデルとして扱う.

ソフトウェアコード部分は, C 言語をモデル内に直接記述し, コード生成に用いた.

また,SW PIM と同じくモデルは UML の要素である.

4.3.7 LIM(Language Independent Model)

LIM(言語独立モデル) は, 特定のハードウェア環境, 特定のソフトウェア環境に依存した UML モデルである.

LIM を構成する UML の要素は,UML のパッケージの Classes（本提案では主に構成を表現する）, Activities（操作に対する振舞いを表現する）からなる.

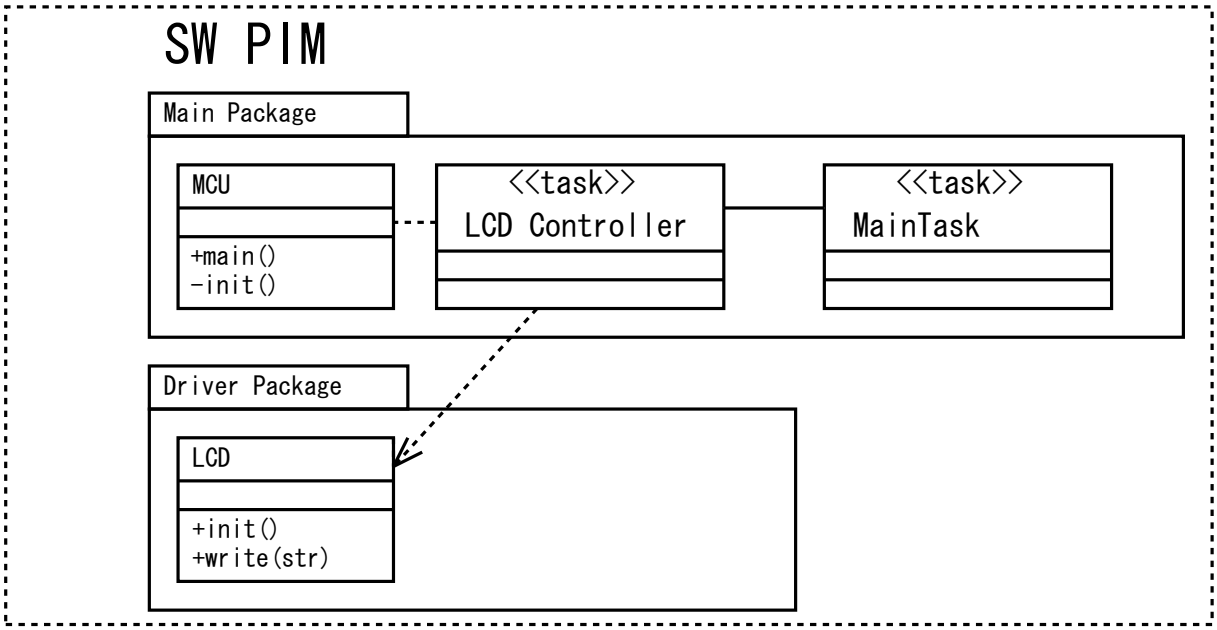

図 4.10: SW PIM モデル例

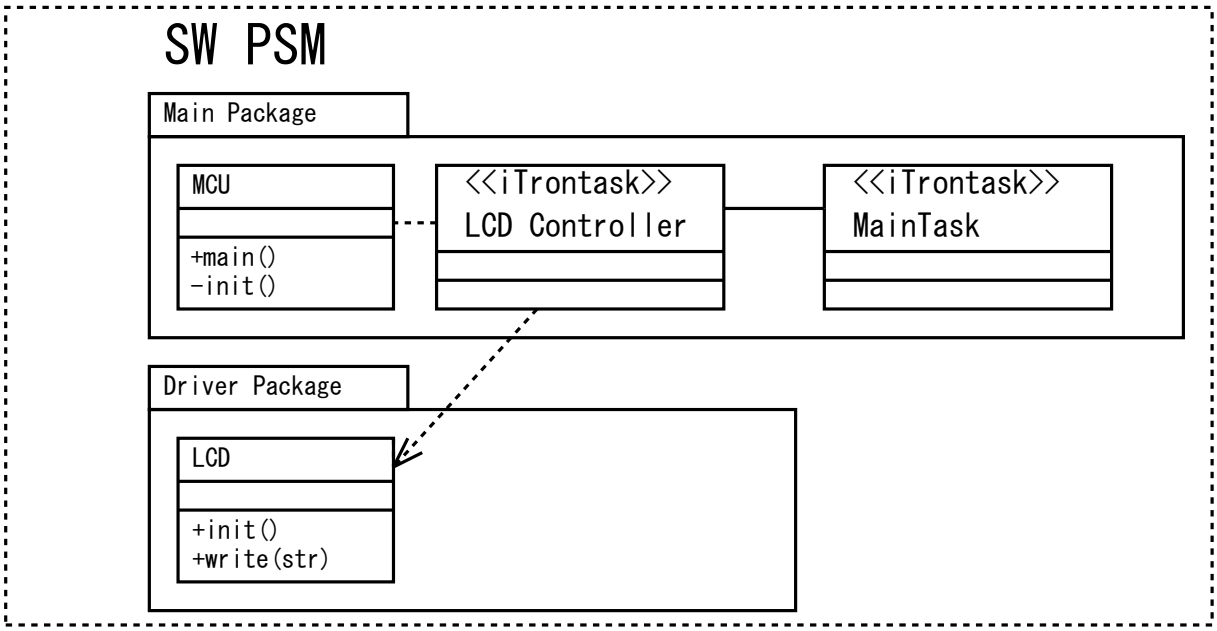

図 4.11: SW PSM モデル例

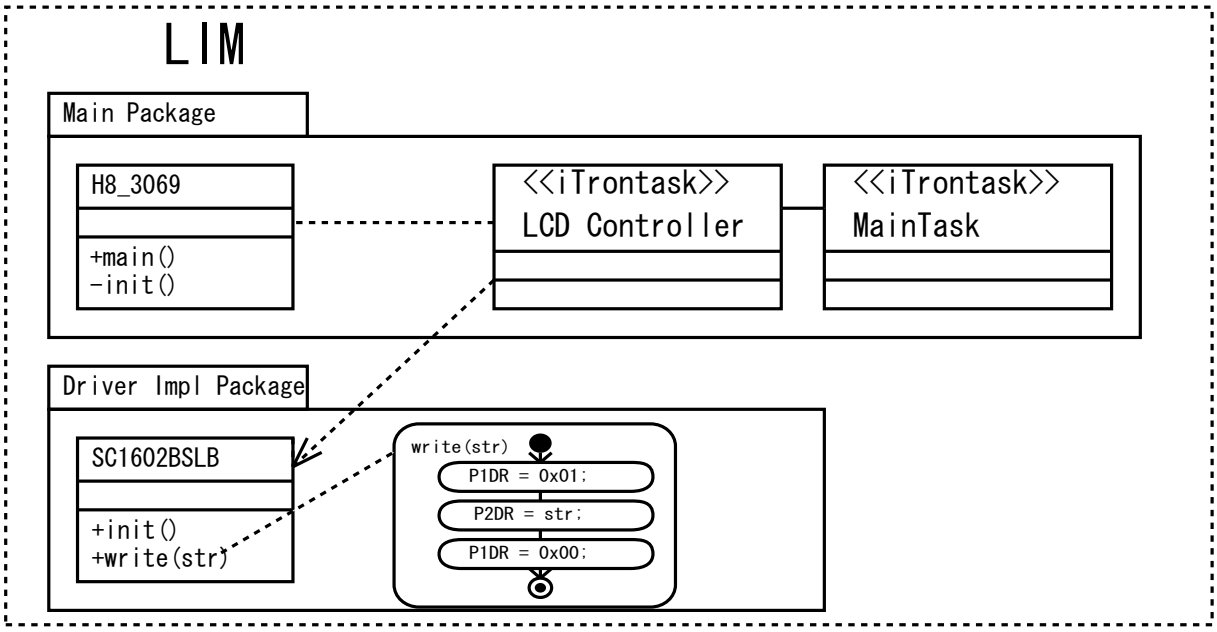

図 4.12: LIM モデル例

4.4 モデル変換

ここでは, 本提案で行う各モデル間の変換について述べる.

4.4.1 HW PIM から HW PSM を生成する

DDM Category の階層構造に従い, HW PIM(親要素) から子要素を特定することで, HW PSM を生成する.

主な変換の操作は,

- HW PIM の要素から, DDMM Category を参照し, 選択可能な子要素を取り出す.
- 取り出した子要素から,HW PSM に適用する要素を選択する.
- 選択された要素を HW PSM に追加する.

生成された HW PSM では,HW PIM では抽象デバイスであったものが特定の MCU,Device に決定される為,DDM structure の要素を参照することが出来る.

これにより, そのデバイスの持つポートやピンの情報を参照し, デバイス間の結線をより詳細に定義することが可能となる.

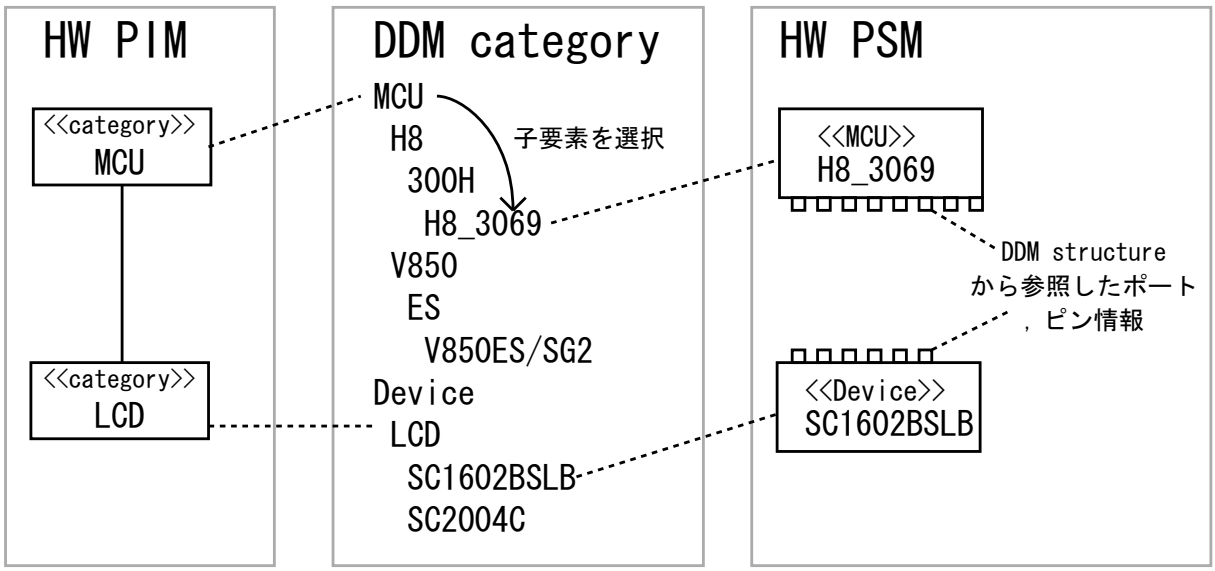

図 4.13: HW PIM からの HW PSM の生成

4.4.2 HW PIM から SW PIM(stub) を生成する

DDM Structure より, デバイスのインターフェイス部分のモデルの生成を行う.
主な変換の操作は以下の通りである.

- MCU 要素は Main パッケージにクラスとして追加する.
- MCU クラスには main(),init() 関数が追加される.
- Device 要素は Driver パッケージにインターフェイスとして追加する.
- Device インターフェイスには,Device 要素が保持している function 要素をメソッドとして追加する.

ソフトウェアのモデルの実装は,Main パッケージ内に追加していき, デバイスを操作する必要がある場合には, そのデバイスのインターフェイスに対して操作を行う.

4.4.3 HW PSM と SW PSM から LIM を生成する

HW PSM と SW PSM を元に DDM の参照を行い, システムのソフトウェアすべてをモデル化した LIM の生成を行う.
ここで行う主な操作は

- HW PSM の結線情報より,MCU のポート等の初期化を行うアクティビティを生成する.

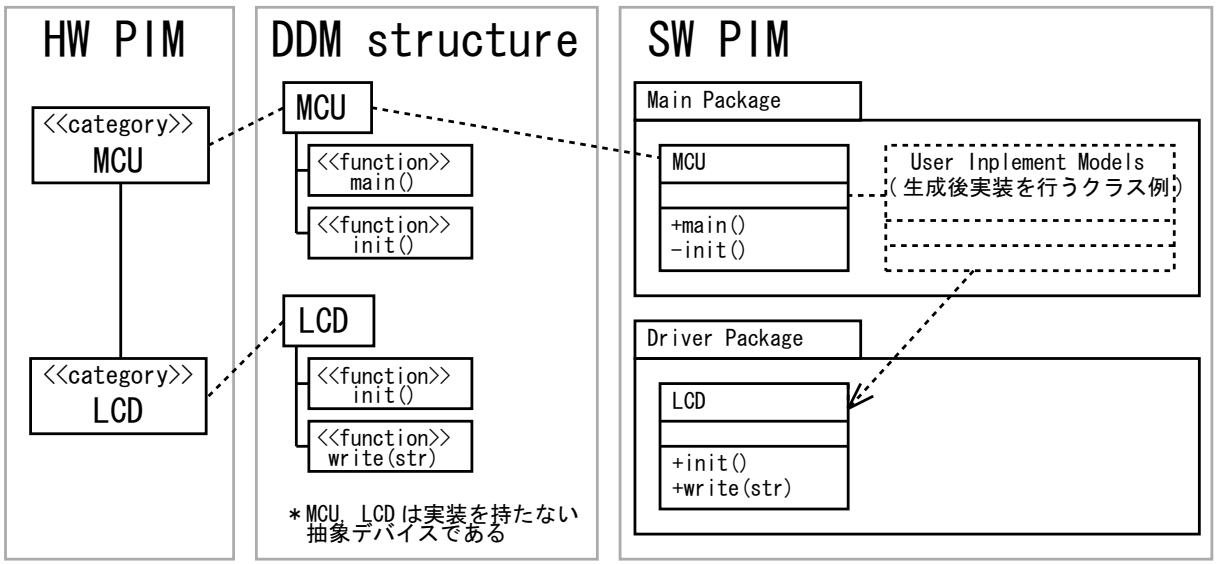

図 4.14: HW PIM からの SW PIM(stub) の生成

- HW PSM の結線情報と DDM behavior で定義されている操作のアクティビティを用いて, device 主体で定義されているアクティビティを,MCU を主体としたアクティビティに変換・生成する. （この部分が, デバイスのインターフェイスに対する実装されたデバイスドライバとなる.）
- 上記で生成された各モデルと SW PSM で作成されたソフトウェアモデルとのマージを行い, LIM を生成する.

である.

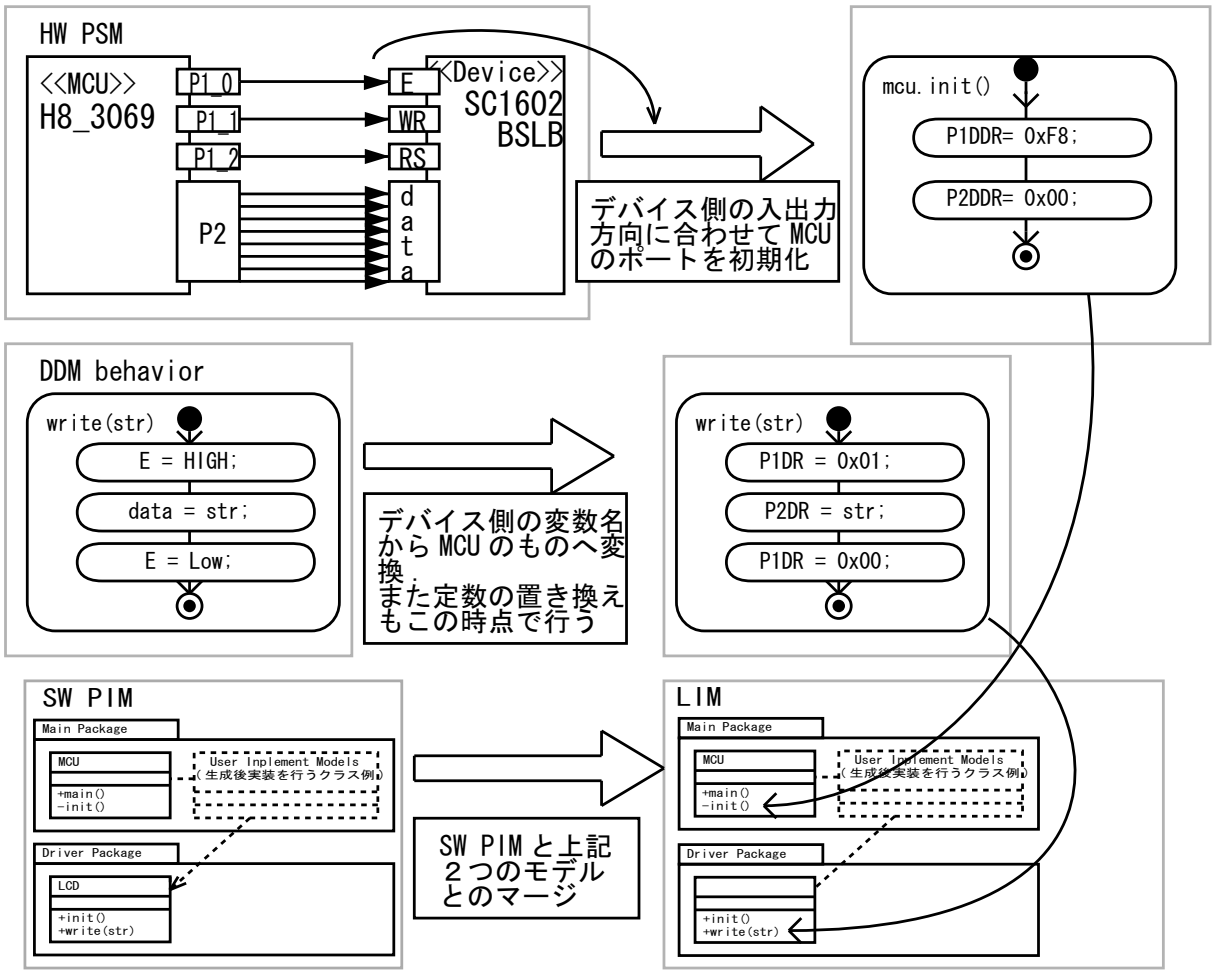

図 4.15: HW PSM と SW PSM からの LIM の生成

4.4.4 LIM からのコード生成

LIM からのコードの生成は, 現時点では C 言語を対象としている. LIM は構造と振舞いの定義された UML であるので, Java,C++等にも変換のルールを付与すれば変換は可能である.

C 言語への変換に関しては,

- １クラスを１ファイルとして変換する
- フィールドは構造体, メソッドは関数として変換する
- 公開するフィールド, メソッドはヘッダファイルに追加する
- レジスタ・ポート等のソフトウェア外で変更が行われる可能性のある変数は volatile キーワードを付加し, コンパイラによる最適化を抑制する.

現時点では, 動的なオブジェクトの生成等に関しては対象としない.

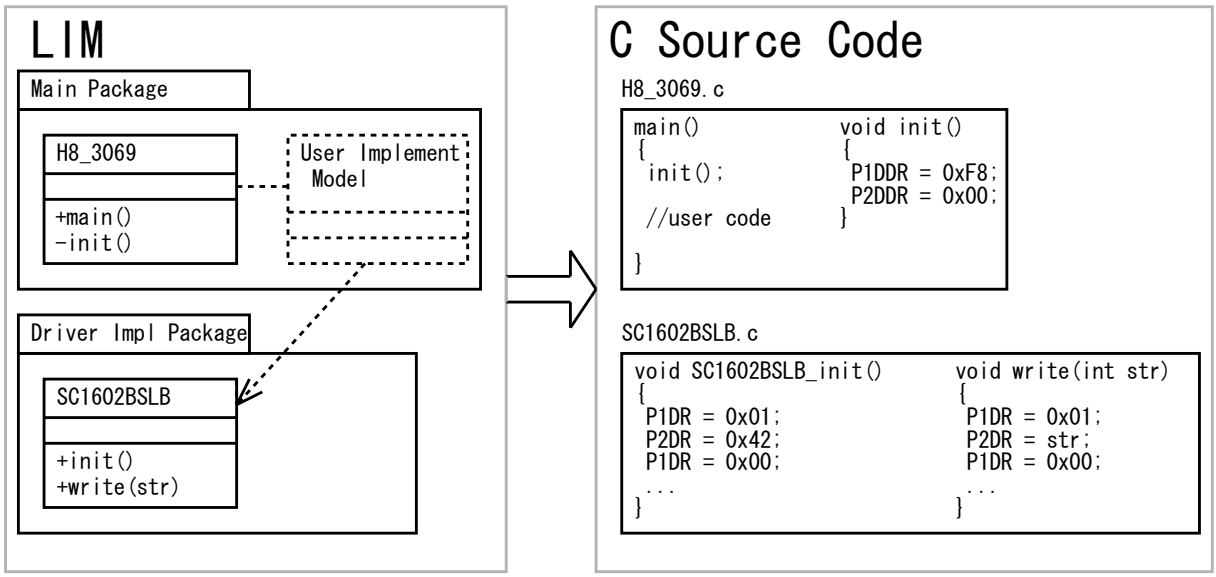

図 4.16: LIM からのコード生成

第5章　システム構成

本章では, 提案手法を元に実装を行ったシステムについて述べる.

5.1　概要

提案手法の評価を行う為に実装を行った.

実装環境は, オープンソースであり非常に強力な開発環境の Eclipse を対象とし Plugin の形で実装を行った. 多くの Eclipse 上の既存技術を利用することで, 効率的に作業を進めることができた.

以下に実装を行った箇所を示す.

- DDMM の各メタモデルの実装
- HW PIM を作成する為のエディタ
- HW PIM → HW PSM 変換
- HW PIM → SW PIM 変換
- HW PSM, SW PIM → LIM への部分的な変換
- LIM からの部分的なコード生成
- 全体の流れを総括するフロントエンド GUI

また作成したシステムは, ソフトウェア名として組込みシステム開発における銀の弾丸を目指す意として Solder Bullet と命名した.

5.2　Eclipse と Eclipse Plugin について

Eclipse はオープンソースの統合開発環境 (IDE) である. 2001 年に IBM が自社の開発環境として開発したものをオープンソースコミュニティに寄付されたもので, オープンソースでありながら商用環境に劣らない高機能を提供する.

Eclipse は一般に java の開発環境であるとの認識が多いが, java の開発環境は Eclipse の一側面に過ぎず, 非常に汎用的で強力な開発環境を提供するものである.

Eclipse は Plugin という形で様々に機能拡張を行うことができ, java の開発環境である JDT も Plugin の一つである. ソフトウェア開発に有用な様々な Plugin が開発されており, 本システムで用いる EMF,oAW なども Plugin の形で提供されている.

また,Plugin 自体を開発することも可能で, 開発との親和性が高いということで, 本システムも Plugin の形で実装を行った.

5.3 システム構成

以下に本システムの全体的な構成を示す.

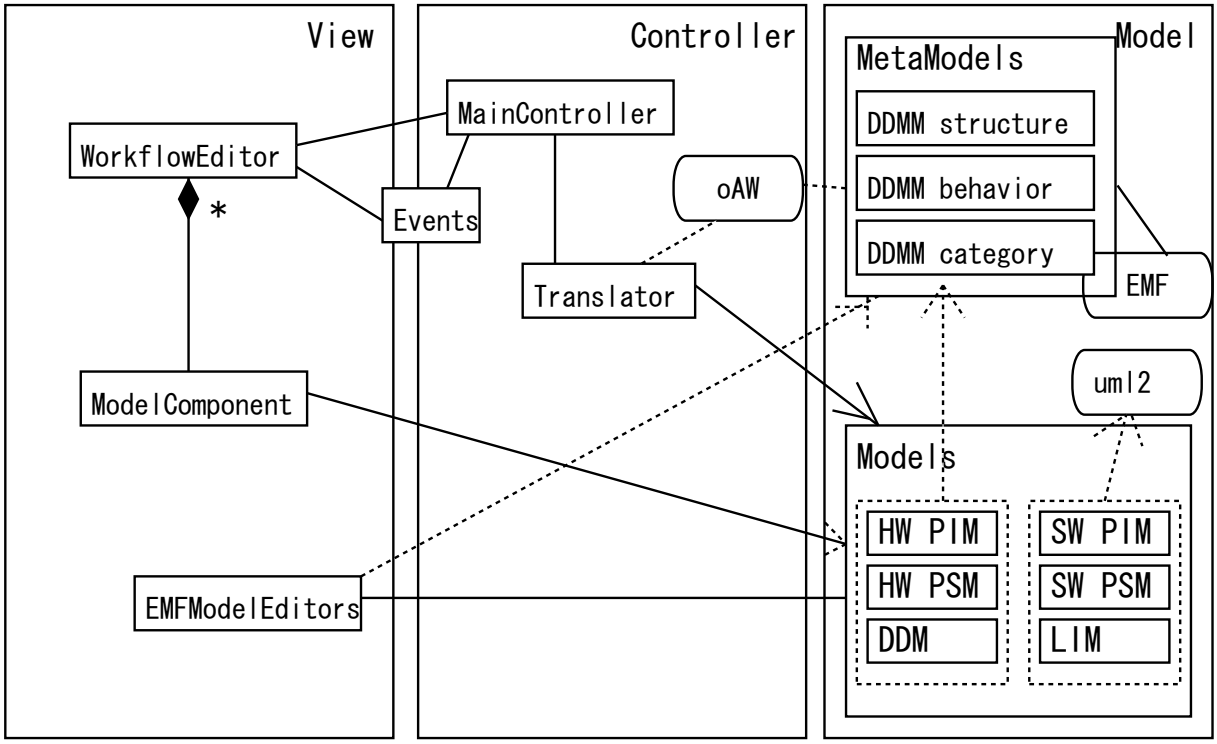

図 5.1: システム構成

5.4 モデル

本システムで用いるモデル･メタモデルは, Eclipse のモデリングフレームである EMF(Eclipse Modeling Framework) を用いて定義している.

EMF は,OMG の提唱する MDA に対して,Java で実装を行ったもので, MOF 実装である ecore, UML2.0 の実装である uml2 などを包括する.

EMF は非常に高性能なフレームワークで, XML や Java のインターフェイス,UML などを入力としてメタモデルを定義することができる. 定義したメタモデルからは Java コー

ドとして実装されたモデルコードを自動生成することができる. また, モデルコードだけではなく, モデルを操作するのに有用なエディットコード, 簡易なモデリングエディタを提供するエディターコードなども, メタモデルの情報だけで自動生成することができる.

本システムでは, 提案で述べた DDMM の category, structure, behavior それぞれのメタモデルの定義を行った.

5.5 モデル変換

EMF は十分に強力な MDA フレームワークであるが, モデル変換に関しては, モデルコードやエディットコードを用いてモデル毎にプログラム内から制御しなくてはならない.

本システムでは, さらに EMF をラップする形で提供されている, oAW(open Architecture Ware) という Plugin を用いた.

oAW は, コンポーネント的に MDA を取り扱うことができ, 変換や, 直列化といった機能を提供するコンポーネントを繋ぎ合わせることで任意の MDA を実現することができる.

本システムでは oAW とは異なったフロントエンドを提供するため, モデル変換に関する部分にのみ oAW を用いた.

5.6 GUI コントローラの実装

上記のモデルの実装とモデル変換の実装だけであっても, MDA を行うことは不可能ではないが, 操作が非常に煩雑であり使いやすいツールとはならない.

これを解消するため全体の流れを俯瞰できる GUI の実装を行った. また実質この部分がシステム全体をコントロールする形となる.

GUI の実装については SWT,jface を用いて, 提案の箇所で示したモデルの流れを模したものを作成した.

ここから各モデルの作成やモデル変換, コード生成などの操作を行う. 現在はまだすべての機能を用いることはできないが, 見通しの良いシステムを構築することができた.

5.7 画面例

以下に本システムを使用している際の画面例を示す.

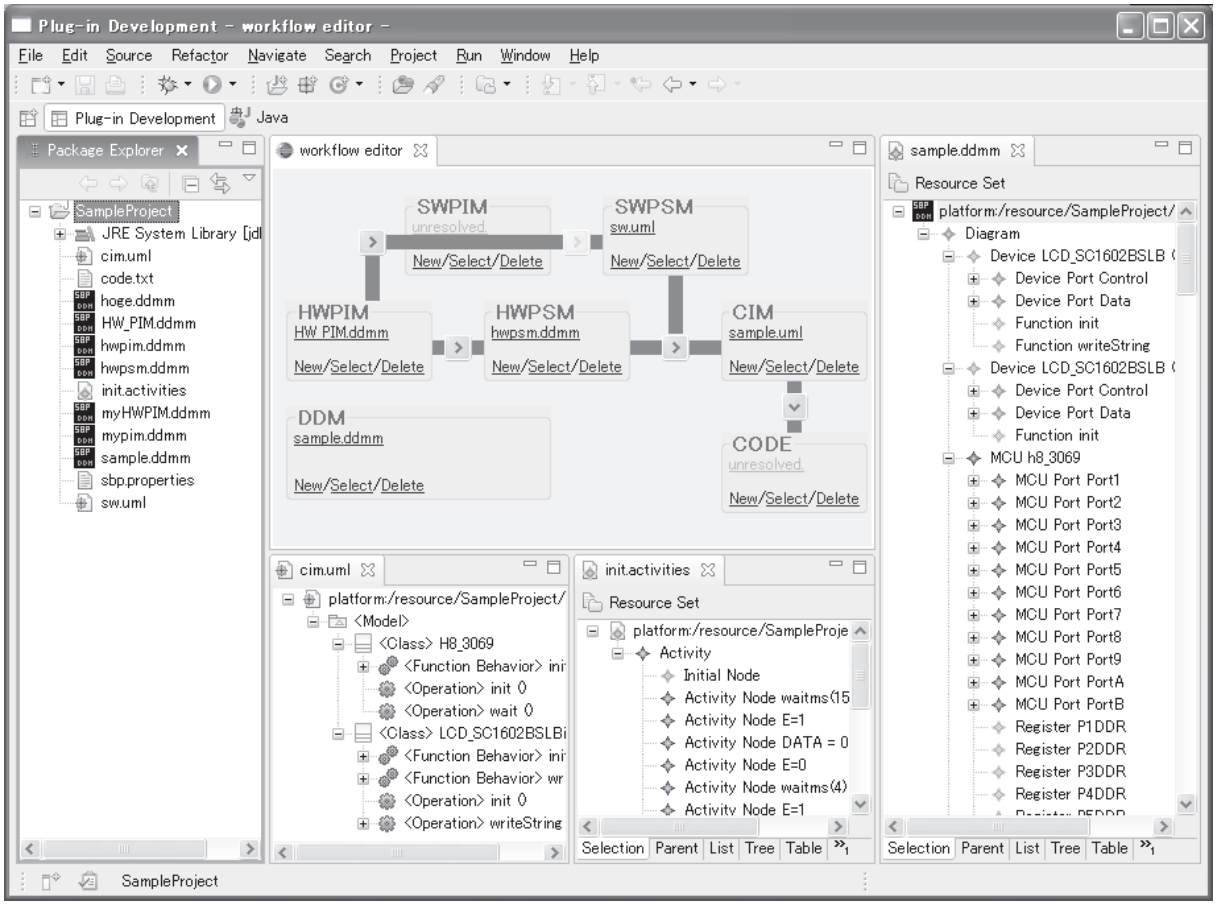

図 5.2: 画面例

第6章　適用例

実装を行った開発環境を用いて, 簡易デジタル時計を例に提案手法の適用を行った.

6.1　簡易デジタル時計

LCD を用いて時刻表示を行うデジタル時計. 表記は 24 時間表記とし, 時刻合わせ等の機能は有さないとする.

ハードウェアは MCU と LCD で構成されており, クロックは MCU 内部のものを用いる.

以下にデジタル時計の外観と想定するハードウェア構成を示す.

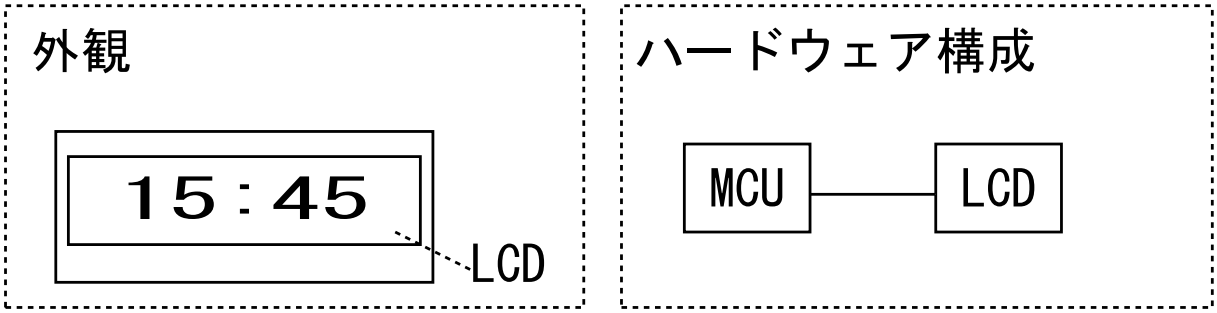

図 6.1: 簡易デジタル時計の外観とハードウェア構成

また, デバイスの例として,

- MCU
  - H8/3069 ： ルネサステクノロジ, 16bit シングルチップマイクロコンピュータ
  - V850ES/SG2 ： NEC エレクトロニクス, 32bit シングルチップマイクロコンピュータ
- LCD
  - SC1602BSLB(8 線モード) : 16 文字× 2 行 キャラクタ表示 超ハイコントラスト LCD モジュール
  - SC1602BSLB(4 線モード) : 上記と同様だが, 配線数と操作法が異なる.

を用いた.

6.2　各モデルとモデル変換

提案手法を適用し,作成したモデルとモデル変換時の動作の詳細について述べる.

6.2.1　DDM

DDM では,デバイス例の各 MCU と LCD のモデル化を行った. 以下にモデル化を行った情報と作成したモデルを図式化したものを示す.

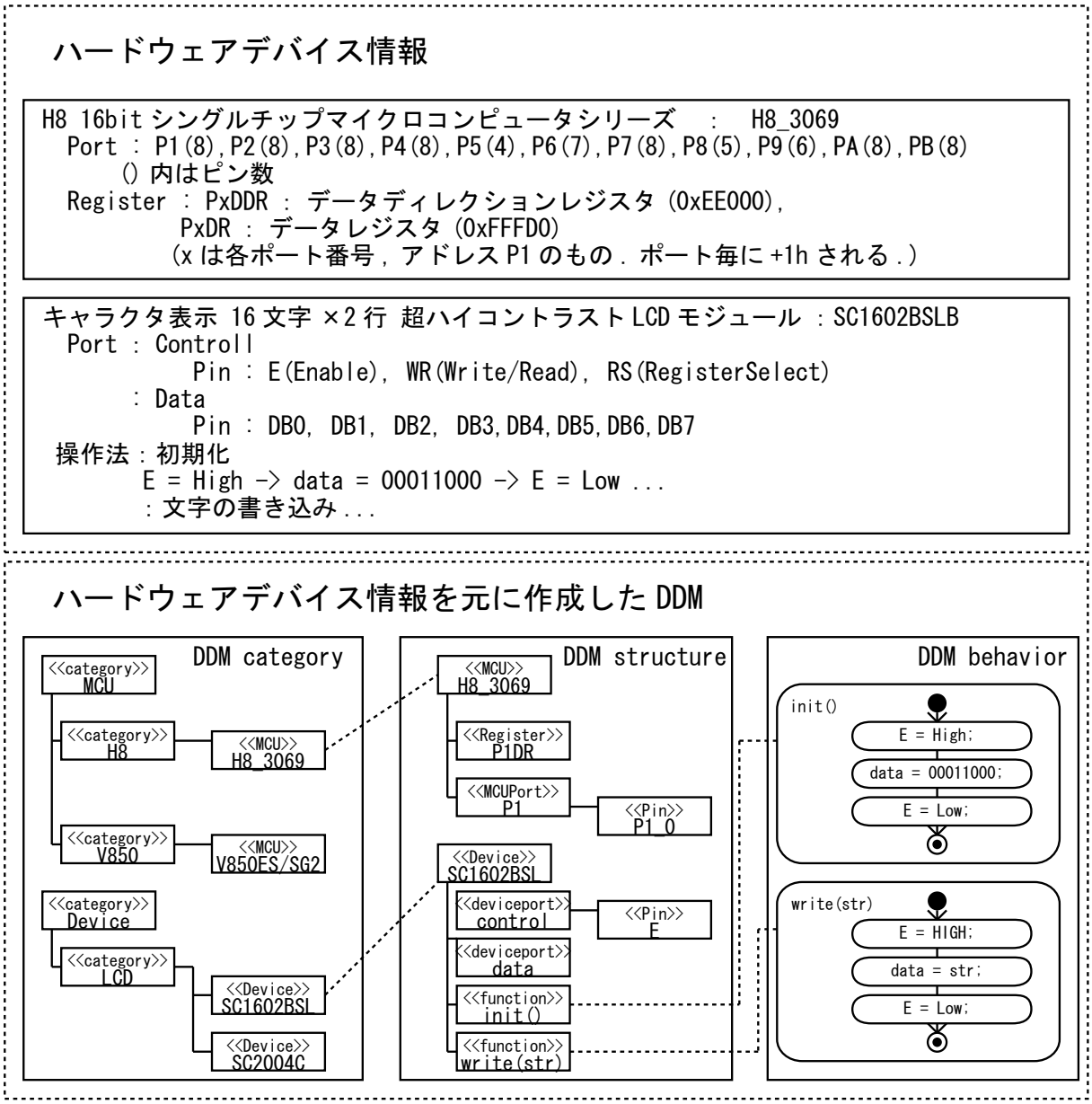

図 6.2: DDM 作成例

また, 以下ににツール上での表示例を示す.

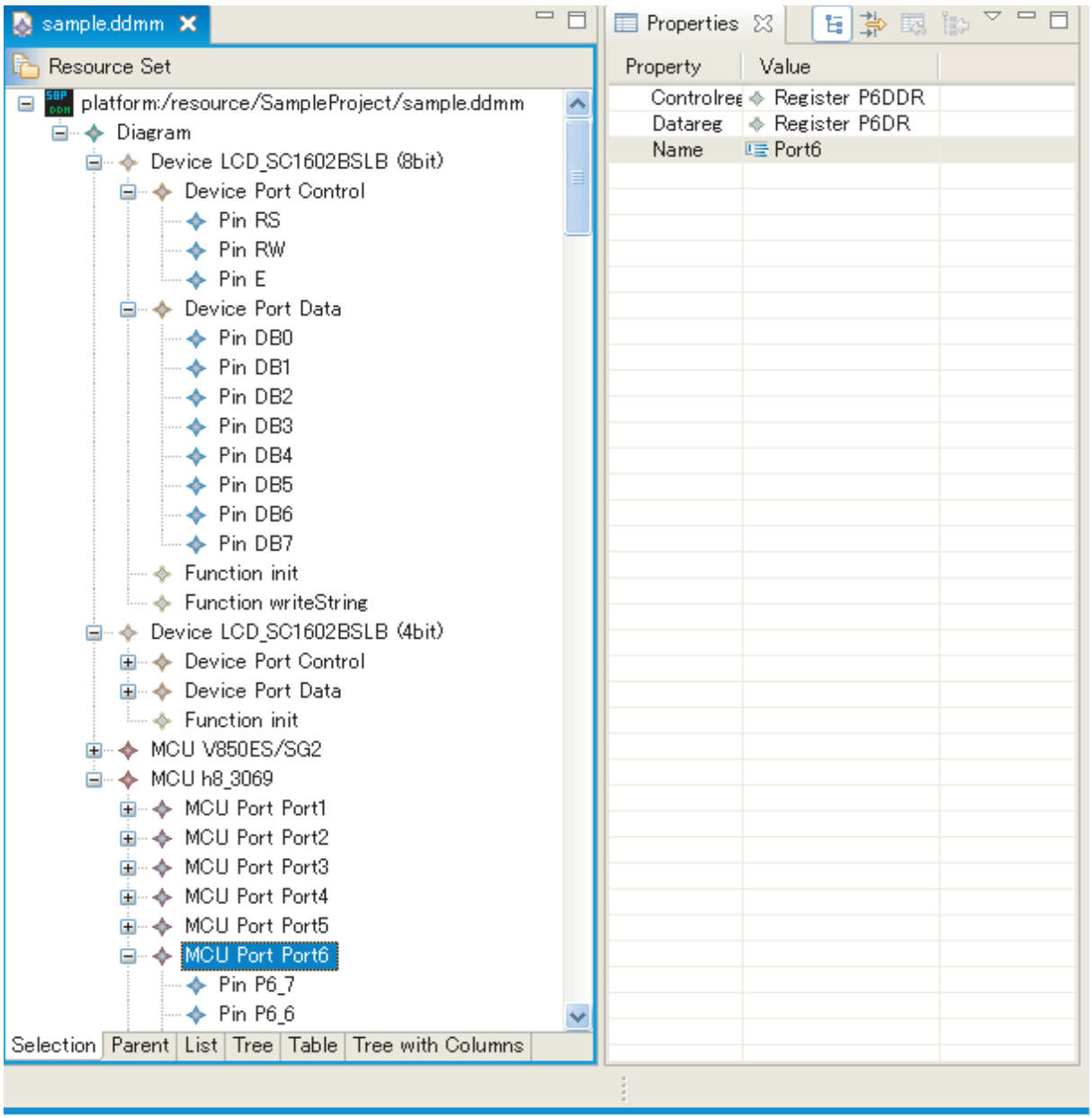

図 6.3: Solder Bullet 上での作成例

6.2.2 HW PIM

以下に 6.1 で示したハードウェア構成を元に HW PIM 作成した.
図は 6.1 のハードウェア構成と同様となる. またツール上での表示は次項に示す.

6.2.3 HW PIM → HW PSM

HW PIM から HW PSM への変換は, 選択ビューにて行う.
以下に操作例を示す.

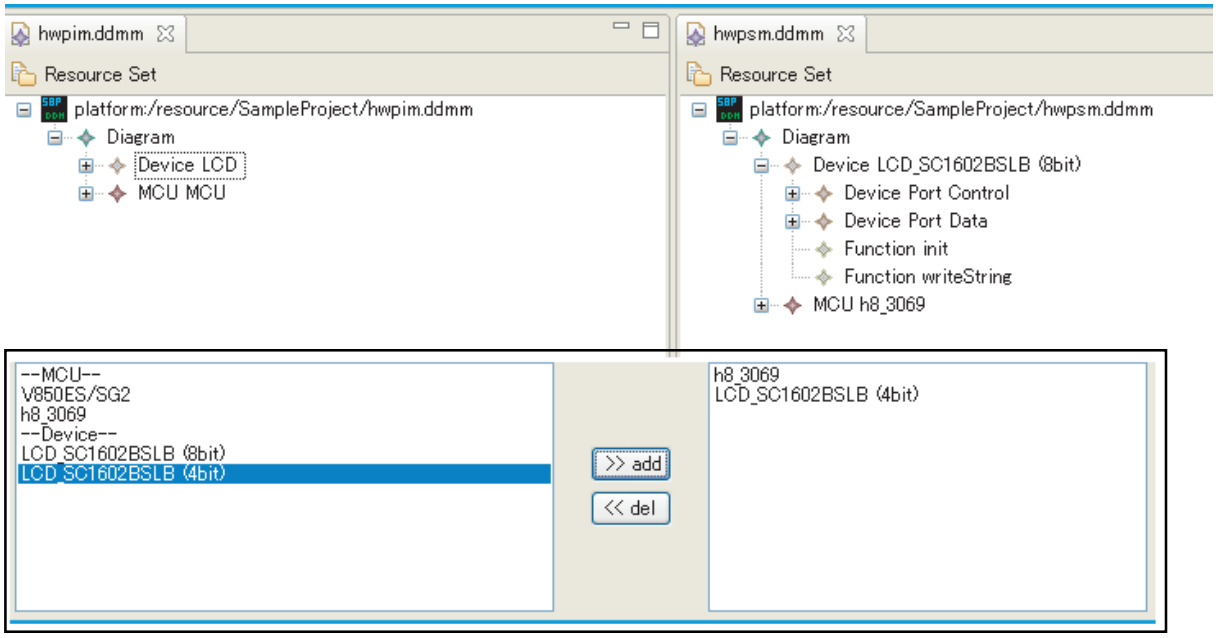

図 6.4: HW PIM(左上),HW PSM(右上) モデルと, 選択ビュー (下囲い)

内部では,DDMcategory を参照し, 表示している.

6.2.4 HW PIM → SW PIM(stub)

HW PIM から SW PIM スタブの生成する.
以下に生成された SW PIM の表示例を示す
内部では,HW PIM を走査し含まれる各モデルに従って uml 要素を作成し,SW PSM に追加している.

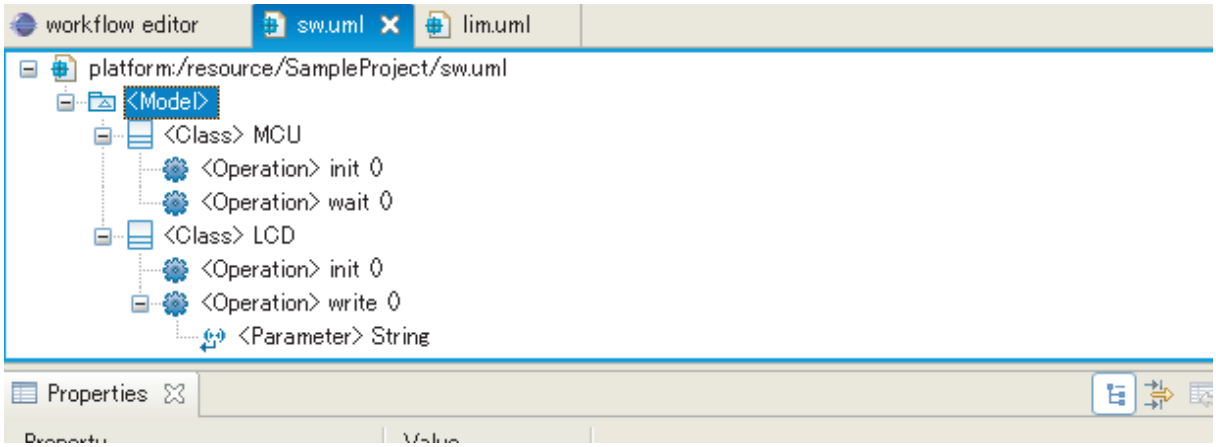

図 6.5: 生成された SW PSM スタブ

6.2.5 HW PSM + SW PIM → LIM

HW PSM と SW PIM からの LIM の生成は, まだ未完成であるため手動でモデルの操作を行い, 生成されうるものを作成した.

以下に作成した LIM を示す.

作成中の機能ではあるが, 内部的には,HW PSM を操作し MCU 要素と Device 要素を判別し, 結合されているポートの情報を元に MCU のデータディレクションレジスタを参照し, 初期化アクティビティの生成を行う.

また,Device 側の Pin を用いて定義されていたものを, 上と同様結合されているポート情報より, MCU 側の Pin に関連付けられているデータレジスタのものに置き換える.

最後に SW PIM のモデルとのマージを行い,LIM を作成する.

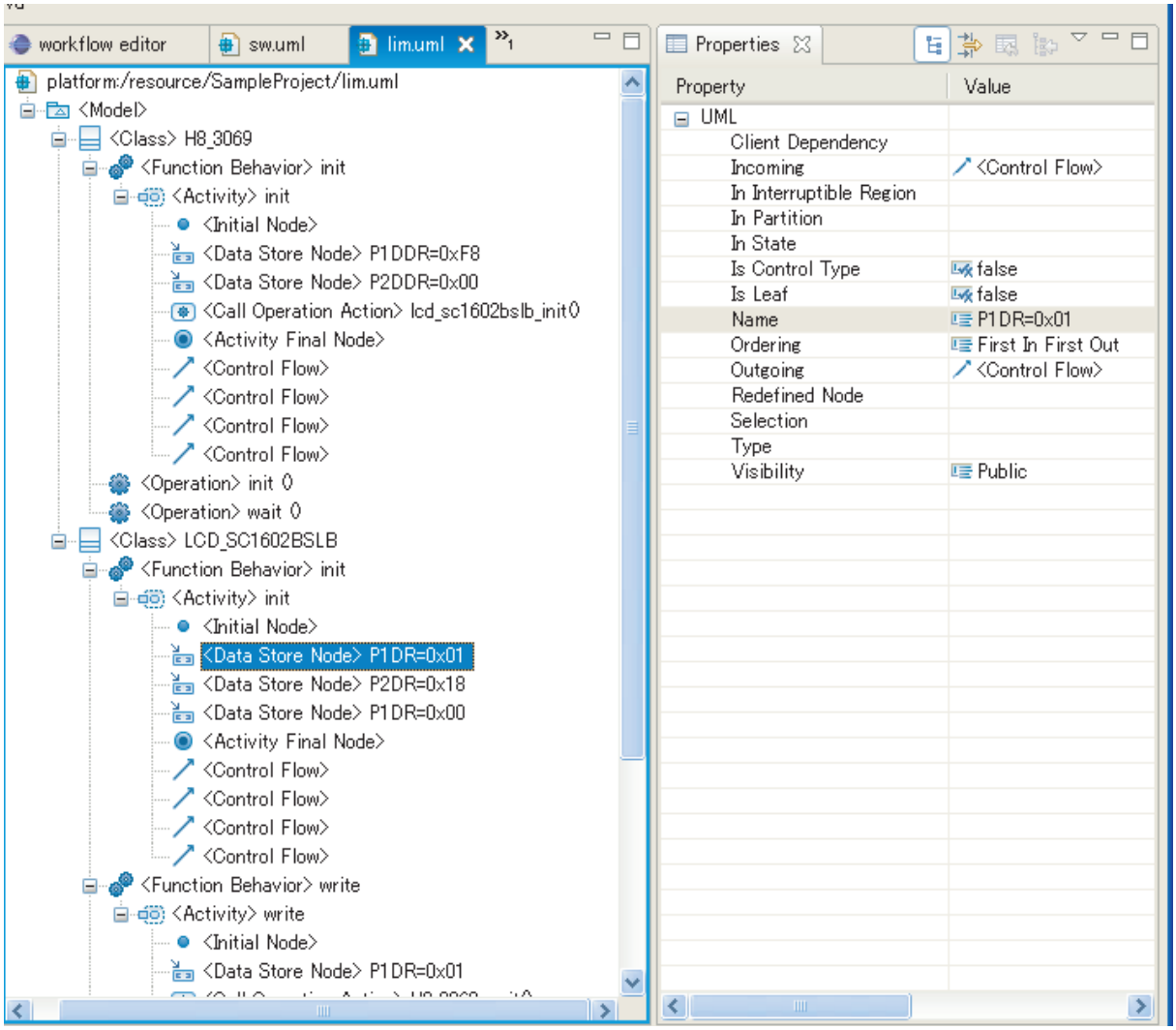
workflow editor
sw.uml
lim.uml
platform:/resource/SampleProject/lim.uml
<Model>
<Class> H8_3069
<Function Behavior> init
<Activity> init
<Initial Node>
<Data Store Node> P1DDR=0xF8
<Data Store Node> P2DDR=0x00
<Call Operation Action> lcd_sc1602bslb_init0
<Activity Final Node>
<Control Flow>
<Control Flow>
<Control Flow>
<Control Flow>
<Operation> init 0
<Operation> wait 0
<Class> LCD_SC1602BSLB
<Function Behavior> init
<Activity> init
<Initial Node>
<Data Store Node> P1DR=0x01
<Data Store Node> P2DR=0x18
<Data Store Node> P1DR=0x00
<Activity Final Node>
<Control Flow>
<Control Flow>
<Control Flow>
<Control Flow>
<Function Behavior> write
<Activity> write
<Initial Node>
<Data Store Node> P1DR=0x01
Properties
Property
Value
UML
Client Dependency
Incoming
<Control Flow>
In Interruptible Region
In Partition
In State
Is Control Type
false
Is Leaf
false
Name
P1DR=0x01
Ordering
First In First Out
Outgoing
<Control Flow>
Redefined Node
Selection
Type
Visibility
Public

図 6.6: 作成した LIM

6.2.6 LIM → Source Code

LIM からのコード生成は, oAW のコード生成エンジンを用いて LIM から C 言語への変換の定義を行った.

SW PIM のモデルと LCD の DDM behavior モデルの実装が不十分であるため, 簡易なコードしか生成することが出来なかったが, 以下に部分的に生成されたコードの例を示す.

```
main(){
      init();

}

init(){
      P1DDR = 0x00;
      P2DDR = 0x00;
      lcd_sb1602bslb_init();
}

lcd_sb1602bslb_init(){
      P1DR = 0x01;
      P2DR = 0x18;
      P1DR = 0x00;
}

lcd_sb1602bslb_write(int data){
      P1DR = 0x01;
      P2DR = data;
      P1DR = 0x00;
}
```

図 6.7: 生成されたコード例

第7章　まとめ

本章では, 提案した手法と開発したシステムについての評価と考察, 今後の課題について述べる.

7.1　評価

提案したハードウェア情報を含めた MDA について, 実装を行い例題を適用した.

ハードウェア情報のモデル化については, 構造面のモデル化は体系立ててまとめることができた. しかしながら振舞いのモデル化は, UML のアクティビティ図の要素を用いて行ったがまだ綺麗にモデル化できているとは言い難い.

各モデルの配置 (HW PIM/PSM, SW PIM/PSM, DDM, LIM) は, 組込みシステムを開発する上での流れにうまく沿う形を形成できた. ハードウェアデバイスのシリーズ構成を抽象ハードウェアとして扱うことで, 様々な段階でのハードウェア構成を扱うことができるようになった.

また特に MDA を用いることで, ハードウェアのカスタマイズ時に, モデルを変更した後のコード生成までの流れをほぼ自動化できることにより, 効率的に開発が行える.

今回は最小限かつ部分的な例題を行っただけであるが, ハードウェア情報のモデル化, ハードウェア構成のモデル化, それらからの UML スタブモデルやコードの生成など, 従来では扱われなかった箇所について, 組込み MDA の可能性を広げることができた.

7.2　考察

通常ソフトウェア開発では取り入れられなかったハードウェア情報をモデル化し MDA に組込むことで, 従来は手動で行っていた部分のコード生成が行えることを確認することができた.

今回は, 開発環境としては実装を行うことができなかったが, ハードウェア情報のモデル化は十分に有益であると思われ, 今後も実装を続け組込みシステム開発に役立てていきたい.

7.3 今後の課題

今後の課題として, まず, 今回完成させることの出来なかったシステムを完全な開発環境として作り上げることが挙げられる.

またSW PIM/SW PSMについても今回は実装を見送ったが, OS等との兼ね合いも含めて今後取り入れて行きたい.

今回は特にコード生成に有効なハードウェア情報のモデル化を行ったが, パフォーマンスの面や, 制約といった側面からも活用できる可能性がある. 仕様書のすべての情報を体系的にモデルに取り入れ, 有効に用いることができれば, 組込みシステム開発をより効率的に進めることができるだろう.

さらに, ハードウェア情報をモデル化したものを各プロジェクト, さらにはネットワークを通じて共有することができればハードウェア情報のモデル化の作業量までも削減することができより効率的な開発が行えるだろう.

謝辞

本研究を進めるに当たり, 終始熱心にご指導頂き本研究をより良い方向へと導いてくださった岸知二特任教授に深く感謝致します. また, ゼミ等を通して貴重なご助言, ご指導を頂いた片山 卓也教授, 青木利晃 助教授に感謝申し上げます. そして, 研究室において何度も貴重な意見を頂いた博士後期過程の金井勇人氏, 研究生活において励まし合った岸研究室, 片山研究室, デファゴ研究室, 青木研究室の友人達に感謝し, 謝辞とします.

参考文献

[1] Devid S. Frankel, 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, 星雲社,2003

[2] スティーブ・メラー, テクノロジックアート, MDA のエッセンス, 翔泳社,2004

[3] フランク・バディンスキー, ディヴィット・スタインバーグ, エド・マークス, レイモンド・イラーシック, ティモシー・グロース, Eclipse モデリングフレームワーク, 翔泳社,2005

[4] UML2.0 Superstructure Specification, Object Management Group, 2005

Appendix A : Solder Bullet 詳細

A.1 : 環境

本システムの開発は,
Eclipse 3.2.1
上で行った. またシステムの対象とする環境も同様である.
また, 利用した主要な Plugin は以下の通りである

- Eclipse 本体の拡張ポイントやリソース周辺
- SWT, jface, draw2d, GMF
- EMF
- UML2
- openArchitectureWare

A.2 : システムの Plugin 構成

以下に本システムの Plugin 構成を示す. 大項目は Plugin を, 子項目はパッケージを表す.

- org.dyndns.junkmiyu.sbp.overview
  システム全体を総括する plugin, 主に overview Editor の実装を行っている.
  - org.dyndns.junkmiyu.sbp.overview.events
    GUI より発行されるイベント類
  - org.dyndns.junkmiyu.sbp.overview.logics
    システム全体のコントロールを行うパッケージ, MVC の C に相当.
  - org.dyndns.junkmiyu.sbp.overview.translator
    各種モデル変換を行う為のクラス類, 内部的に oAW のモデル変換を用いている.
  - org.dyndns.junkmiyu.sbp.overview.views
    GUI を構築するコンポーネント類

  - org.dyndns.junkmiyu.sbp.overview.wizards
    モデルの新規作成ウィザード等

- org.dyndns.junkmiyu.sbp.ddmManager
  DDM の管理を GUI から行う為の plugin, 現在未実装

- org.dyndns.junkmiyu.sbp.ddmreader
  DDM の入力を補佐する plugin, CVS 形式のデータから DDM 要素へ変換する.

- org.dyndns.junkmiyu.sbp.hwpimEditor
  HW PIM モデルを GUI で操作する為の plugin

  - org.dyndns.junkmiyu.sbp.hwpimEditor.editor
    エディタの実装を含むパッケージ

- org.dyndns.junkmiyu.sbp.hwpsmEditor
  HW PSM モデルを GUI で操作するための plugin, 現在未実装

- org.dyndns.junkmiyu.sbp.mm.ddmm __ behavior
  DDMM behavior メタモデルを実装した plugin, メタモデルの定義を含む

  - org.dyndns.junkmiyu.sbp.mm.ddmm __ behavior
    メタモデルの Java 実装.EMF による自動生成

- org.dyndns.junkmiyu.sbp.mm.ddmm __ behavior.edit
  ddmm __ behavior パッケージより生成された EditPlugin

- org.dyndns.junkmiyu.sbp.mm.ddmm __ behavior.editor
  ddmm __ behavior パッケージより生成された EditorPlugin

- org.dyndns.junkmiyu.sbp.mm.ddmm __ category
  DDMM category メタモデルを実装した plugin, メタモデルの定義を含む

- org.dyndns.junkmiyu.sbp.mm.ddmm __ category.edit
  ddmm __ category パッケージより生成された EditorPlugin

- org.dyndns.junkmiyu.sbp.mm.ddmm __ category.editor
  ddmm __ category パッケージより生成された Editor Plugin

- org.dyndns.junkmiyu.sbp.mm.ddmm __ structure
  DDMM structure メタモデルを実装した plugin, メタモデルの定義を含む

- org.dyndns.junkmiyu.sbp.mm.ddmm __ structure.edit
  ddmm __ structure パッケージより生成された EditPlugin

- org.dyndns.junkmiyu.sbp.mm.ddmm __ structure.editor
  ddmm __ structure パッケージより生成された EditorPlugin