

| | |
|--------------|---|
| Title | ハードウェアモデルを用いたモデル駆動型ソフトウェアプロダクトライン |
| Author(s) | 細合, 晋太郎 |
| Citation | |
| Issue Date | 2015-03 |
| Type | Thesis or Dissertation |
| Text version | ETD |
| URL | http://hdl.handle.net/10119/12753 |
| Rights | |
| Description | Supervisor:Defago Xavier, 情報科学研究科, 博士 |

博士論文

ハードウェアモデルを用いた
モデル駆動型システムプロダクトライン

細合 晋太郎

主指導教員 DEFAGO Xavier

北陸先端科学技術大学院大学
情報科学研究科

平成27年3月

目次

| | | |
|----------|--|----------|
| 1 | 序論 | 1 |
| 2 | 背景知識 | 3 |
| 2.1 | 本論文における表記法 | 3 |
| 2.2 | 組込みシステム | 4 |
| 2.2.1 | 組込みシステムの特徴 | 5 |
| 2.2.2 | 組込みシステムの構成 | 5 |
| 2.2.3 | 組込みシステム開発 | 6 |
| 2.3 | 組込みソフトウェア | 7 |
| 2.3.1 | 組込みソフトウェアの特徴 | 7 |
| 2.3.2 | 組込みソフトウェアの構成 | 8 |
| 2.3.3 | 組込みソフトウェア開発 | 9 |
| 2.4 | プロダクトライン開発 | 11 |
| 2.4.1 | ソフトウェアプロダクトライン開発の活動 | 11 |
| 2.4.2 | フィーチャモデル | 12 |
| 2.4.3 | 組込み製品のシステムプロダクトライン開発 | 13 |
| 2.5 | モデル駆動開発 | 13 |
| 2.5.1 | モデリング | 14 |
| 2.5.2 | UML | 14 |
| 2.5.3 | Model Driven Architecture(MDA) | 14 |
| 2.5.4 | MDA の4層アーキテクチャ | 15 |
| 2.5.5 | MDD と UML | 15 |
| 2.5.6 | ドメインモデル | 16 |
| 2.5.7 | モデル変換 | 17 |
| 2.6 | ドメイン特化言語 | 17 |

| | | |
|----------|--|-----------|
| 2.6.1 | 内部 DSL と外部 DSL | 17 |
| 2.6.2 | グラフィカル DSL とテキスト DSL | 18 |
| 2.7 | 組込みシステム向けモデリング言語 | 18 |
| 3 | 問題 | 21 |
| 3.1 | 本研究の想定する組込みシステムプロダクトライン開発 | 21 |
| 3.2 | 組込みシステムプロダクトライン開発における問題 | 22 |
| 3.3 | ゲートウェイドライバ導出における問題 | 23 |
| 3.3.1 | ハードウェアの多様性 | 24 |
| 3.3.2 | ハードウェア情報の多様性 | 24 |
| 3.3.3 | 開発手順の暗黙性 | 25 |
| 4 | アプローチ | 27 |
| 4.1 | 全体像：組込みシステム向けモデル駆動システムプロダクトライン 開発 | 27 |
| 4.2 | システムプロダクトラインによる構成管理 | 29 |
| 4.2.1 | 段階的なフィーチャモデル導出のプロセス | 29 |
| 4.2.2 | ハードウェアフィーチャモデル | 30 |
| 4.2.3 | ソフトウェアフィーチャモデル | 30 |
| 4.2.4 | ハードウェアフィーチャとソフトウェアフィーチャの依存関係 | 31 |
| 4.2.5 | 2つのフィーチャモデルの定義と利用 | 31 |
| 4.3 | ゲートウェイドライバの導出 | 32 |
| 4.3.1 | ゲートウェイドライバ導出のプロセス | 32 |
| 4.3.2 | 入出力モデルの形式化 | 33 |
| 4.3.3 | 入力情報 DSL | 33 |
| 4.3.4 | ハードウェアモデルからゲートウェイモデルまでの変換定義 | 34 |
| 5 | 提案手法 | 36 |
| 5.1 | システムプロダクトライン | 36 |
| 5.1.1 | 2層フィーチャモデルのメタモデル定義 | 36 |
| 5.1.2 | 2層フィーチャモデルの利用 | 38 |

| | | |
|----------|---------------------------------------|-----------|
| 5.2 | ゲートウェイドライバ導出 | 40 |
| 5.2.1 | ゲートウェイドライバの導出プロセス | 40 |
| 5.2.2 | ハードウェア情報の選択 | 41 |
| 5.2.3 | ハードウェア情報とゲートウェイドライバの形式化 | 45 |
| 5.2.4 | ハードウェア情報入力用 DSL | 49 |
| 5.2.5 | ハードウェアモデルの結合 | 60 |
| 5.2.6 | ゲートウェイドライバ導出の変換定義 | 61 |
| 6 | 評価 | 67 |
| 6.1 | 提案手法の実装 | 67 |
| 6.1.1 | 実装環境 | 67 |
| 6.1.2 | 2層フィーチャメタモデルの実装 | 68 |
| 6.1.3 | ハードウェアメタモデル, ゲートウェイメタモデルの実装 | 69 |
| 6.1.4 | DSLの実装 | 69 |
| 6.1.5 | モデル変換の実装 | 69 |
| 6.1.6 | コード生成の実装 | 69 |
| 6.1.7 | モデル駆動型システムプロダクトライン開発環境としての統合 | 70 |
| 6.2 | 例題システム | 71 |
| 6.3 | 提案手法の適用 | 72 |
| 6.3.1 | システムプロダクトライン | 72 |
| 6.3.2 | ゲートウェイドライバの導出 | 74 |
| 6.4 | 評価 | 86 |
| 6.4.1 | 組み込みシステムプロダクトライン開発における問題 | 87 |
| 6.4.2 | ゲートウェイドライバ導出における問題点 | 88 |
| 7 | 関連研究 | 90 |
| 7.1 | フィーチャモデルと製品導出 | 90 |
| 7.1.1 | フィーチャモデルの分割 | 90 |
| 7.1.2 | 段階的な製品導出 | 91 |
| 7.2 | デバイスドライバの生成 | 92 |
| 7.3 | プロダクトライン開発とモデル駆動開発 | 93 |

| | | |
|----------|----------------|-----------|
| 8 | 議論 | 95 |
| 8.1 | 貢献 | 95 |
| 8.2 | 適用性 | 96 |
| 8.3 | 再利用性 | 97 |
| 9 | 結論 | 98 |
| 9.1 | 結論 | 98 |
| | 謝辞 | 100 |
| | 本研究に関する研究業績 | 106 |

目 次

| | | |
|------|--|----|
| 2.1 | 表記法 | 3 |
| 2.2 | 組込みシステムの構成例 | 6 |
| 2.3 | 組込みソフトウェアの構成 | 8 |
| 2.4 | フィーチャモデル | 12 |
| 2.5 | MDA の 4 層アーキテクチャ | 15 |
| 2.6 | 従来開発とモデル駆動技術による導出 | 16 |
| 2.7 | MARTE のパッケージ構造 | 19 |
| 2.8 | AUTOSAR 下位ソフトウェアレイヤ | 20 |
| 3.1 | ハードウェア情報 | 25 |
| 4.1 | アプローチの全体像 | 28 |
| 4.2 | 段階的なフィーチャモデルの導出プロセス | 29 |
| 4.3 | ゲートウェイドライバの導出 | 32 |
| 5.1 | 2 層フィーチャモデルのメタモデル | 37 |
| 5.2 | 2 層フィーチャモデルを用いた製品導出のプロセス | 38 |
| 5.3 | 制約されたソフトウェアフィーチャモデルの導出 | 39 |
| 5.4 | ゲートウェイドライバの導出プロセス | 40 |
| 5.5 | 拡張した MARTE のパッケージ構造 | 46 |
| 5.6 | ハードウェア部品 DSL の定義 | 51 |
| 5.7 | ハードウェア部品 DSL (MCU) の定義 (一部抜粋) | 51 |
| 5.8 | ハードウェア部品 DSL (MCU) の入力例 | 53 |
| 5.9 | ハードウェア部品 DSL (MCU) の DSL からハードウェア部品モデルへの変換 | 54 |
| 5.10 | ハードウェア部品 DSL (デバイス) の定義 (一部抜粋) | 54 |

| | | |
|------|---|----|
| 5.11 | ハードウェア部品 DSL (デバイス) の入力例 | 55 |
| 5.12 | ハードウェア部品 DSL (デバイス) の DSL からハードウェア部品 モデルへの変換 | 56 |
| 5.13 | ブロック図 DSL の定義 | 56 |
| 5.14 | ブロック図 DSL の入力例 | 57 |
| 5.15 | ブロック図 DSL からブロック図モデルへの変換 | 57 |
| 5.16 | 回路図 DSL の定義 | 58 |
| 5.17 | 回路図 DSL の入力例 | 59 |
| 5.18 | 回路図 DSL から回路図モデルへの変換 | 60 |
| 5.19 | ハードウェアモデルの結合 | 60 |
| 5.20 | ハードウェアモデルからゲートウェイモデルへの変換の概要 | 61 |
| 5.21 | デバイスドライバの導出 | 62 |
| 5.22 | ペリフェラルドライバの導出 | 64 |
| 5.23 | MCU 初期化処理の導出 | 65 |
| 5.24 | ゲートウェイモデルからゲートウェイドライバのコード生成 | 66 |
| 5.25 | コードテンプレート例 | 66 |
| 6.1 | 統合環境の実行画面 | 70 |
| 6.2 | 例題システムの製品系列 | 71 |
| 6.3 | アプリケーション例 | 72 |
| 6.4 | ハードウェアフィーチャモデル | 73 |
| 6.5 | ソフトウェアフィーチャモデル | 73 |
| 6.6 | ソフトウェアフィーチャとハードウェアフィーチャの依存関係 | 74 |
| 6.7 | 導出されたハードウェアプロダクトと, 制約されたソフトウェア フィーチャモデル | 74 |
| 6.8 | データシート | 76 |
| 6.9 | ハードウェア部品 DSL | 77 |
| 6.10 | ハードウェア部品モデル | 78 |
| 6.11 | ブロック図と入力したブロック図 DSL | 79 |
| 6.12 | ブロック図モデル | 79 |
| 6.13 | 回路図 | 80 |

| | |
|------------------------------------|----|
| 6.14 ネットリスト | 81 |
| 6.15 回路図 DSL | 82 |
| 6.16 回路図モデル | 83 |
| 6.17 ハードウェアモデル | 84 |
| 6.18 ゲートウェイモデル | 85 |
| 6.19 生成されたゲートウェイドライバのコード | 86 |

表 目 次

| | | |
|-----|----------------------|----|
| 5.1 | 主要なメタモデル要素 | 48 |
| 6.1 | 使用環境 | 68 |
| 6.2 | ハードウェア部品一覧 | 75 |

第 1 章

序論

組込みシステムは何らかの機器に組み込まれ、その機器の監視や制御を行うコンピュータシステムである。近年マイクロプロセッサ技術の進展と機器の高機能化などに伴い、組込みシステムの重要性が一層増し、より大規模かつ高品質の組込みシステムを短期に開発することが求められるようになってきている。多くの組込みシステムでは多品種開発を行っているため、プロダクトライン開発 [4] に注目が集まっている。しかしながら従来のプロダクトライン開発は、ソフトウェア部分のプロダクトライン開発（ソフトウェアプロダクトライン開発）が中心で、ソフトウェアが依存するハードウェアに関する情報は取り扱うことが難しかった。このためソフトウェアに加えハードウェアのバリエーションも製品系列として分析するシステムプロダクトライン開発 [32] に注目が集まっている。

システムプロダクトライン開発を実現するためには、ハードウェア、ソフトウェアそれぞれの製品系列を捉えるだけでなく、その間のインタフェースを扱うことが重要となる。ソフトウェアからハードウェアにアクセスするインタフェースは、ハードウェアの仕様に基づいて実装されるが、従来の開発ではインタフェース部分はアドホックに作られることが多く、システムプロダクトライン開発の中でインタフェース部分を体系的に導出することは困難であった。

そこで本研究では、システムプロダクトライン開発を実現するために、ハードウェアとソフトウェアのフィーチャの依存関係を考慮した上で整理し、ハードウェアフィーチャから体系的に導出することを目的とする。

本論文の構成は以下のとおりである。2 章で研究の基礎となる背景知識について

述べる．3章で組込みシステムプロダクトライン開発における問題点の分析を行い，4章で問題を解決するためのアプローチを述べる．5章ではアプローチに基づいた提案手法を述べ，6章で提案手法の実装並びに適用評価を行う．7章では関連研究との対比し，8章で本研究に関わる技術的な議論を行い，9章で総括する．

第 2 章

背景知識

2.1 本論文における表記法

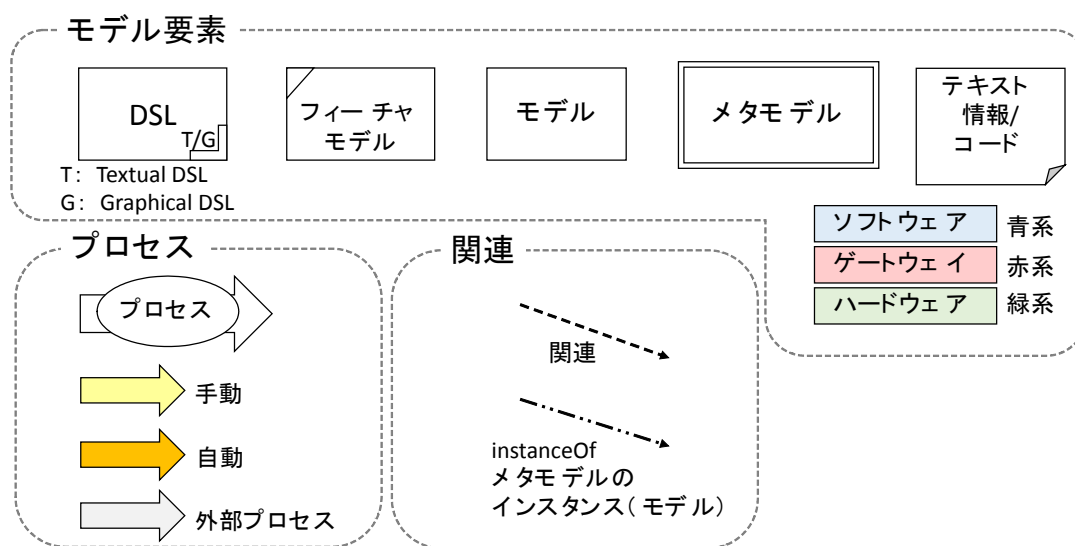


図 2.1: 表記法

本論文の表記法について示す。本論文では多くのモデル図を用いる。モデル図においては図の表記方に基いて記載している。記法は要素の形状と色で表す。右下に「」がついたものは DSL，左上に三角があるものはフィーチャモデル，四角の要素はモデル，二重四角のものはメタモデルを表す。右下が折れた要素はテキスト要素を表す。またこれらの形状に加え，青系の色はソフトウェア，赤系は後述す

るゲートウェイ，緑系はハードウェアに関する要素であることを示す。

モデル間の変換や入力といったプロセスは大矢印で示し，矢印に付与した楕円の内部にプロセス名を記述する．また，色によってプロセスの種類を区別する．黄色のプロセスは，人の手による手動プロセス，橙はプログラムやアプリケーションによる自動プロセス，灰色は今回は対象としない外部のプロセスを表す。

要素間の関連は矢印で表す．破線は依存関係を示し，適宜テキストにて関連名を付与する．二点破線は矢印元の要素が矢印先のメタモデルに沿うことを示す。

2.2 組込みシステム

組込みシステムは何らかの機器に組み込まれ，その機器の監視や制御を行うコンピュータシステムである．近年マイクロプロセッサ技術の進展と機器の高機能化などに伴い，組込みシステムの重要性が一層増し，より大規模かつ高品質の組込みシステムを短期に開発することが求められるようになってきている．組込みシステムといっても，携帯電話，家電，オーディオ，車載システムなど多岐に渡る．携帯電話などでは近年共通のハードウェアプラットフォームが利用されるようになってきたが，家電，オーディオ，車載システムなどの中小規模の組込みシステムでは，製品に応じてハードウェアプラットフォームが変化する場合も多い。

このような組込みシステムでは，製品によってハードウェア構成が異なり，ハードウェアの仕様に合わせてソフトウェアの開発を行う必要がある．本研究ではこのようなハードウェアの共通プラットフォームを持たず，ハードウェア構成に合わせてソフトウェアを開発する必要のある組込みシステムを対象とする。

このような組込みシステム開発特有の難しさとして，ハードウェア構成が変わることが多く，それに合わせてソフトウェアも作り直さねばならないことが多い点を指摘できる．特にソフトウェアとハードウェアのインタフェース部分はハードウェア構成に極めて強く依存するため，システム開発において重要となる。

2.2.1 組込みシステムの特徴

組込みシステムの特徴として，リアルタイム性や省電力性，高信頼性，高品質性などが挙げられるが，組込みシステム開発に関するものとして，本研究は以下の2つに注目する．

- 生産コスト

本研究の対象とする中小規模のソフトウェアは相対的に短い期間での開発が必要となるが，共通プラットフォームがないため，ソフトウェアだけでなく，ハードウェアやそれとのインタフェース部分を含めて開発する必要があり生産コストを圧迫する．そのため体系だった開発，再利用，あるいは自動化などが求められる．

- ハードウェアとソフトウェアの相互設計

製品ごとにハードウェアの構成が変わることが多く，それにあわせてソフトウェアを開発しなければならない．ハードウェア依存の部分には特有のスキルが必要となり開発を一層難しくしている．

2.2.2 組込みシステムの構成

組込みシステムは，一般的にソフトウェアの搭載されるMCU（Micro Control Unit）と複数のデバイスから構成される．ここでMCUとは，CPUやメモリと様々な機能の周辺機能（ペリフェラル:Peripheral）をワンチップ化したコンピュータシステムである．また，本研究ではMCUから制御されるハードウェア部品一般をデバイスと呼ぶ．

図2.2にMCUとデバイス間の関係の詳細を示す．MCUとデバイス間は電氣的に接続されると共に何らかの通信方法に則って接続される．通信方法にはI²CやSPI，USARTといったシリアル通信の他，汎用IOを信号線として用いた独自の通信方法やアナログ値を扱うものもある．通信方法は多くの場合MCUに内蔵される周辺機能によってハードウェアとしてプロトコルが実装されており，MCUとデバイス間はこの周辺機能を介して接続される．

ソフトウェアはCPU上で動作し、周辺機能の提供する通信機能を用いてデバイスを制御する。しかしながらソフトウェアからは周辺機能を直接参照することができず、MCU内のレジスタを介して、接続されている周辺機能を制御してデバイスと通信を行う。そのためMCUのレジスタを介して周辺機器を制御するためのドライバ(ペリフェラルドライバ)を用意し、デバイスを制御するドライバ(デバイスドライバ)は、ペリフェラルドライバを介してデバイスを制御する構成をとる。なおペリフェラルドライバもデバイスドライバの一部に含め全体をデバイスドライバと呼ぶこともあるが、本稿では周辺機能を制御するソフトウェアをペリフェラルドライバ、MCU外のデバイスを制御するソフトウェアをデバイスドライバと呼び、両者を合わせてハードウェアを制御するソフトウェアをゲートウェイドライバとする。

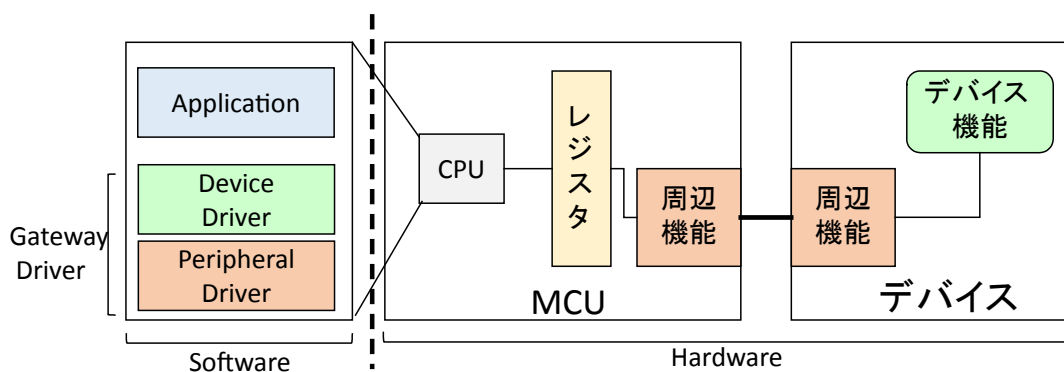


図 2.2: 組み込みシステムの構成例

2.2.3 組み込みシステム開発

組み込みシステム開発では、まずシステム全体の要件定義を行う。またこの際製品ファミリを展開するものや、前製品を引き継ぐものであれば、共通部分や可変部分の分析、再利用資産の構築についての検討などを行う。

続いて、ハードウェア、ソフトウェアの切り分けを行う。この際ハードウェア、ソフトウェアのインタフェースが重要となる。ハードウェア、ソフトウェアを切り分けそれぞれの役割を明確化した後、ハードウェア、ソフトウェアそれぞれの開発を行う。開発の際は互いの仕様を考慮した上で設計を行うが、特にソフトウェア

はハードウェアに強く依存するため、ハードウェアとソフトウェアのバリエーションを考慮した上で、ソフトウェアとハードウェア間の依存関係を適切に管理することが重要である。

2.3 組込みソフトウェア

2.3.1 組込みソフトウェアの特徴

組込みソフトウェアの特徴としても、高信頼性、省リソース、リアルタイム制約などがあげられる。組込みソフトウェアは、機器に組み込まれその制御を行うソフトウェアであるため、不具合により機器に損害を与える危険性や、人に危害を加えてしまう危険性を内包している。このため組込みソフトウェアには高い信頼性が求められる。

組込みシステムは量産品であることが多いため、ハードウェアのコストが重要視される。このため、ソフトウェアを可能な限り軽量に作り少ないメモリで動作することが求められる。また、消費電力についても制約が加わることも多く、ソフトウェアにおいても少消費電力となるよう考慮する必要がある。

組込みソフトウェアでは、機器の制御や外界とインタラクションするために、多くの場合実世界の時間（リアルタイム）に関わる制約が求められる。例えば、一定時間内に処理を完了させる必要性や、特定の時間ちょうどに制御指示を出すといったことが求められる。

また、組込みソフトウェアは高い品質も求められる、このため実績のあるソフトウェア部品を再利用資産とするような配慮がなされている。

2.3.2 組み込みソフトウェアの構成

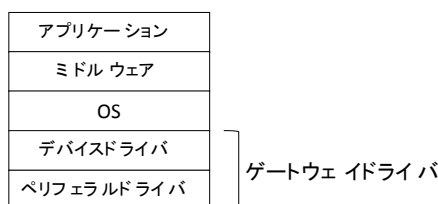


図 2.3: 組み込みソフトウェアの構成

図 2.3 に組み込みソフトウェアの典型的な構成を示す。ソフトウェアの構成は下位のレイヤから順に、ハードウェアを操作するゲートウェイドライバ等の上に OS があり、その上にミドルウェアさらにシステムとして機能を提供するようなアプリケーションが配置される。このソフトウェア構成はあくまで一例であり、実際にはソフトウェアではミドルウェアや OS がない場合などもあるが、製品によって組み合わせは変化するが、基本的にはこうした構成をとるものが多い。

2.3.2.1 デバイスドライバ、ペリフェラルドライバ

多くの組み込みシステムでは、MCU の外部に接続されたデバイスを制御して、ハードウェアや電気回路、機械などの制御を行う。この際、MCU に内蔵されている CPU で直接制御するのではなく、周辺機能を介して制御を行う場合が多い。周辺機能は MCU に内蔵されるハードウェアで通信やタイマ、デジタル・アナログの I/O など様々な機能を提供する。MCU とデバイス間の通信を行う際、多くの場合この通信機能を介してデバイスを制御する。

デバイスドライバは、MCU 外に接続されたデバイスを制御するためのソフトウェアである。MCU とデバイス間は周辺機能を利用して接続される事が多いため、デバイスドライバは上記のペリフェラルドライバを利用する事が多い。(構成によってはアドレスバスにデバイスを接続し、メモリ空間に直接マッピングされるものもある) ゲートウェイドライバはハードウェア制御の詳細を隠蔽し、その上位に構築されるアプリケーションソフトウェアにデバイスや周辺機能のハードウェア機能を提供する API となる。

ゲートウェイドライバはハードウェアに強く依存するため、その開発にはソフトウェアの知識だけでなくハードウェアに関する知識が求められるため、その開発には特有の難しさがある。

2.3.2.2 Operating System(OS)

OS は、タスクスケジューリング機能や、ファイルシステムを提供するソフトウェアである。特に組込みシステムでは、リアルタイム性を重視することが多いため、Real Time OS (RTOS) が利用される。代表的な RTOS として、OSEK OS[1], uITRON[2], FreeRTOS[3] 等がある。

2.3.2.3 ミドルウェア

ミドルウェアは、特定用途向けの機能を提供するためのソフトウェアパッケージで通信機能やデータベース、GUI、音声処理、画像処理など多くのミドルウェアが提供されている。

2.3.2.4 アプリケーションソフトウェア

アプリケーションソフトウェアは、システム固有のサービスを提供するソフトウェアで、下位のソフトウェアの機能を統合し、サービスを提供する。OS、ミドルウェアはアプリケーションソフトウェアの要求する機能に応じて既成品を選択することが多いが、アプリケーションソフトウェアは製品固有であるため都度作成される事が多い。

2.3.3 組込みソフトウェア開発

本項では、組込みソフトウェアの内、特にハードウェアを制御するゲートウェイドライバの開発について述べる。ゲートウェイドライバの開発は大きく、デバイスドライバ、ペリフェラルドライバ、MCU の初期化処理の 3 つに分けられる。以下それぞれについて述べる。

2.3.3.1 デバイスドライバ

図 2.2 に示したように、デバイスは MCU の持つ周辺機能を介して接続され、周辺機能で実装されているプロトコルを介して制御を行う。このためデバイスドライバは、ペリフェラルドライバを用いてデバイスの要求する制御手順に従って実装される。

デバイスドライバの開発手順として、一般的に以下のような流れが挙げられる。

1. ハードウェアの構成図であるブロック図を参照し、どのようなデバイス、MCU が接続されているかを確認し、接続に用いられるプロトコルを確認する。
2. 対象のデバイスのデータシートを参照し、どのような機能を有するか確認し、その機能を利用するための操作手順（一般にプロトコルのレベルで記載されている）を確認する。
3. 提供するハードウェア機能毎に操作手順に沿って、ペリフェラルドライバを利用し実装する。

2.3.3.2 ペリフェラルドライバ

一般に MCU には、複数の周辺機能が内蔵されており、接続するデバイスやシステムに対する要求に応じて適切なものを選択して実装する。ペリフェラルドライバの開発手順として、以下のような流れが挙げられる。

1. ブロック図を参照し、どの周辺機能が用いられているか確認する。デバイスとの接続に用いる周辺機能以外にもシステムに対する要求によって利用するものがあれば確認する。
2. 回路図とデータシートを確認し、周辺機能の利用する I/O を確認する。競合がある場合は代替手段を検討する。
3. MCU のデータシートの内、対象の周辺機能の項を参照し、利用するための操作手順（一般にレジスタ操作で記載されている）を確認する。
4. 周辺機能の操作手順にしたがい、機能ごとにレジスタの操作列として実装を行う。

2.3.3.3 初期化処理

MCUには複数の周辺機能があり、排他的であったりI/Oが競合する場合がある。このため、周辺機能を選択するための設定や、利用するI/Oに関する設定を行う必要がある。初期化処理の開発方法として、以下が挙げられる。

1. ブロック図とハードウェア仕様を参照し、利用する周辺機能を確認する。
2. 回路図とMCUのデータシートのI/Oの項と周辺機能の項を確認し、周辺機能が利用するI/Oや入出力方向、利用方法を確認する。
3. MCUのデータシートの周辺機能の項を確認し、その周辺機能を利用するための設定を確認し、初期化処理として実装する。
4. MCUデータシートのI/O項を参照し、I/Oの利用方法に応じた設定を確認し、初期化処理として実装する。

2.4 プロダクトライン開発

ソフトウェアの同じような品種を製品の系列として捉え、予めその製品の系列の多様性を考慮した設計を行うソフトウェアプロダクトライン開発 (Software Product Line Development : SPLD)[4] に注目が集まっている。ソフトウェアプロダクトラインはソフトウェアの製品系列を想定し、その共通性と変動性の分析に基づいて、再利用性の高いソフトウェア部品 (コア資産) を作成し、計画的に製品系列を展開することで効率を高めるソフトウェア開発手法である。

2.4.1 ソフトウェアプロダクトライン開発の活動

ソフトウェアプロダクトライン開発には、製品系列の再利用資産となるコア資産を開発するドメインエンジニアリングと、コア資産を組み合わせる個々の製品を開発するアプリケーションエンジニアリングの2つの活動がある。

2.4.1.1 ドメインエンジニアリング

ドメインエンジニアリングでは，製品を展開していく領域（ドメイン）の特徴を分析し，製品系列全体を効果的に開発するためのコア資産（再利用資産）を構築することが主な目的となる．製品系列中に含まれる各プロダクトの共通性や可変性を分析し，各プロダクトにおいて利用されうる再利用可能なソフトウェア部品（コア資産）を開発する．本研究ではこの可変性分析において，各プロダクトの特徴を分析しフィーチャモデルとして体系的に整理することを想定する．

2.4.1.2 アプリケーションエンジニアリング

アプリケーションエンジニアリングではドメインエンジニアリングで作成したコア資産を活用して，実際の製品を作成していく．フィーチャモデルの要素を選択することにより，必要なコア資産の組み合わせを選択し，それに基づきプロダクトを開発する．

2.4.2 フィーチャモデル

フィーチャモデル [19] はソフトウェアの製品系列を分析し，フィーチャ（機能・非機能を含むソフトウェアの特徴）を木構造に整理するためのモデルである．

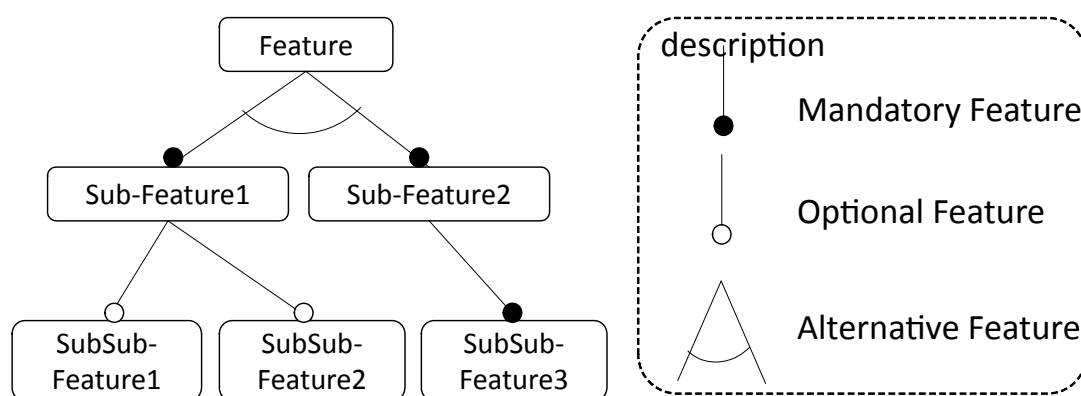


図 2.4: フィーチャモデル

フィーチャモデルの記法を図 2.4 に示す。フィーチャモデルは最上位を根とした木構造となっており、上位ほど抽象度が高く、下位に行くほど詳細なフィーチャとなる。フィーチャモデルではフィーチャの子要素の選択方法に対して様々な制約を加える。Mandatory（黒丸）はその子要素が必須であることを示す。Optional（白丸）はその子要素が選択可能であることを示す。

また、Alternative は複数の子要素に足して指定し、子要素群の中から一つだけ選択されることを示す。

複数のソフトウェアのフィーチャの差異をフィーチャモデルで分析することにより、再利用性の高いソフトウェア部品を特定し、組み合わせにより製品系列を導出できるようにする。

2.4.3 組込み製品のシステムプロダクトライン開発

SPLD では一般にシステム全体を対象として、製品系列の分析を行っていくが、組込みシステムでは、ソフトウェアとは別にハードウェアの製品系列が存在すると捉えた方が組込みシステム開発の工程に沿う場合も多い。

特に本研究で対象とするような組込みシステムでは、ソフトウェアの製品系列とハードウェアの製品系列の両方を扱うため、両者の依存関係が複雑になることが多い。このためソフトウェアプロダクトライン開発に比べてシステムプロダクトライン開発はこうしたソフトウェアとハードウェアの間の依存関係を含めた可変性管理や製品導出が重要となる。

2.5 モデル駆動開発

モデル駆動開発（Model Driven Development:MDD）とは、抽象度の高いモデルを開発の中心としたソフトウェア開発手法である。従来はコード中心の開発が主流であったが、ソフトウェアの様々な観点を抽象化したモデルを開発の中心とすることで、必要な観点を捉え効率的な開発を行う。また、モデルからコードまでの変換を定義しておくことにより、抽象度の高いモデルからコードまでの変換を自動的に行うことができ、生産性の高い開発手法となる。

2.5.1 モデリング

モデリングとは、ソフトウェアを特定の観点と抽象度で表現する作業であり、これにより特定の工程で注目すべき情報のみを取り扱うことができる。ソフトウェアのモデルは大きく構造のモデルと振舞いのモデルの二種類に分けることができ、構造のモデルでは各抽象度のソフトウェアの静的な構造、振舞いのモデルでは様々な観点からソフトウェアの要素間の動的な側面をモデル化する。

2.5.2 UML

UML (Unified Modeling Language) とは、OMG が規定しているソフトウェアをモデリングするための汎用的なモデルセットである。UML の版は現時点で 2.4.1 で、14 のモデル図が規定されている。主なモデルとして、クラス図、オブジェクト図といった構造モデルと、状態マシン図やシーケンス図といった振舞いモデルが規定されている。

2.5.3 Model Driven Architecture(MDA)

Model Driven Architecture(MDA)[5] とは OMG(Object Management Group) の提唱するモデルを中心とした開発フレームワークである。

MDA では、PIM(Platform Independent Model:プラットフォーム独立モデル) と PSM(Platform Specific Model:プラットフォーム依存モデル) を区別して設計を行う。1 つの PIM を設計しておくことで複数の PSM に変換することが可能である。従来の開発サイクルでは、モデルはドキュメントとして用いられ、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。一方 MDA ではこうしたモデルを機械処理してソースコードまでの導出することなどを想定している。機械語からアセンブラ、アセンブラから C 等の高級言語と移ってきたように、モデリング言語を入力とした開発に移ろうとしている。

2.5.4 MDA の 4 層アーキテクチャ

MDA の重要な要素として、4 層のメタモデルアーキテクチャが挙げられる。モデルを利用するためには、そのモデルの定義が必要となるが MDA では、メタモデルと呼ばれるモデルを定義するためのモデルを用いて定義を行っている。前述の UML もメタモデルが規定されている。

メタモデルはさらにメタメタモデルで定義されている。MDA ではこのメタメタモデルとして MOF (Meta Object Facility) [6] などを用いる。なお、メタメタモデルはメタメタモデル自身によって定義されている。このようなメタメタモデルが規定されていることにより、メタメタモデルに沿うメタモデル間の整合性が保たれ、モデル変換の定義や様々なツール間の相互互換が実現されている。

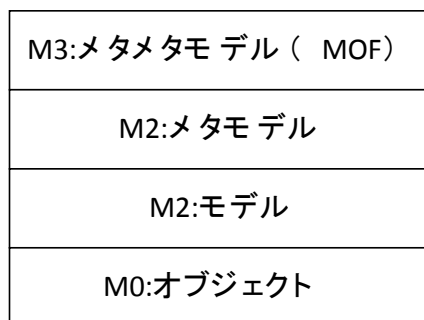


図 2.5: MDA の 4 層アーキテクチャ

2.5.5 MDD と UML

MDD を実現する際、多くの場合 UML を用いる事が多い。OMG では UML のメタモデルを定義している。このため、UML メタモデルを用いた定義を行うことで、UML の各図の要素から別の図要素、プログラミングコードへと変換することができる。

図 2.6 に MDD のプロセスを示す。

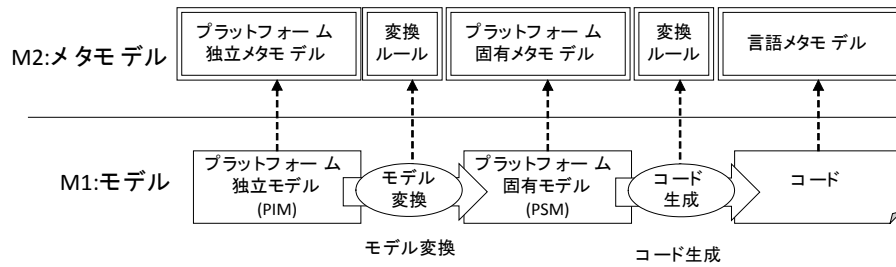


図 2.6: 従来開発とモデル駆動技術による導出

2.5.6 ドメインモデル

MDA とは別に、UML を用いずドメイン固有のモデルを開発の中心に据えるドメイン駆動開発 (Domain Driven Development (DDD)) [8] が注目されている。

UML は汎用性が高いが故に、特定の技術領域 (ドメイン) をモデリングしようとしても、抽象度が高く、ソンドメイン特有の記述が行いづらい。

ドメイン駆動開発では、特定のドメインに限定し、そのドメインでしか使えないが具象的なモデルを定義し用いる。用途を特定することで、モデルとコードの対応関係を詳細に定義することができ、UML を用いる場合に比べ MDD を実現しやすい。

ドメインモデルの定義には大きく 2 つの方法が取られる。一つは UML のプロファイルを用いるもので、UML にステレオタイプ等の付与情報を加えることで、特定ドメイン向けに特化させるものである。ドメインに必要なモデルが UML で表現しやすい場合は、また、汎用の UML エディタを用いることができるメリットもある。

もう一方は、メタメタモデルを利用し独自にドメインモデルのメタモデルを作成する方法である。プロファイルでの定義に比べ、UML のメタモデルに縛られないため柔軟な定義が可能である。汎用の UML エディタでは扱えないため、後述するドメイン特化言語 (Domain Specific Language (DSL)) [7] を用いてモデル入力を行うことが多い。

2.5.7 モデル変換

モデル変換は、あるモデルから別のモデルへの変換、またはモデルからコードへの変換を行う。モデルからモデルへの変換では、メタモデル間の変換規則を定義することでモデル間の変換を行う。上記した PIM から PSM での変換も同様に、PIM のメタモデルから PSM のメタモデルへの変換規則を定義して行う。

モデルからコードへの生成は大きく二種類あり、対象となる言語の言語メタモデルへの変換規則を定義する方法とテンプレートエンジンを用いたものがある。言語メタモデルへの変換規則を用いるものは、上記のモデル間の変換と同様である。テンプレートエンジンを用いるものは、生成したいテキストにモデル要素を埋め込めるようにマークアップしておき、入力したモデルを当てはめることで、コード生成を行う。

2.6 ドメイン特化言語

ドメインモデルは特定のドメインの記述に特化したモデルであるが、同様に特定のドメイン記述に特化しプログラミング言語として使われる記述もある。これらをドメイン特化言語 (Domain Specific Language:DSL) と呼ぶ。

DSL では多くの場合、DSL から実行可能なプログラムコードを生成する。この実行可能なプログラムコードの生成には、モデル-テキスト変換などの技術が多く使われている。DSL は実装方法や表現方法によって以下に述べるようないくつかの種類が存在する。

2.6.1 内部 DSL と外部 DSL

内部 DSL とはプログラム言語内で、言語機能を拡張し、特化言語を作成する DSL である。主にプログラムの特にリフレクションなどの言語機能自体にアクセスする機能を利用し、パーサプログラムを介して独自言語を解釈して、対象のプログラム言語に変換する。

一方、外部 DSL とはプログラム言語に依存せず、独自に特化言語を作成する DSL である。言語定義言語を持つものが多く、Yacc/Lex など外部 DSL を作成するた

めの環境といえる。

2.6.2 グラフィカル DSL とテキスト DSL

テキスト DSL とはテキスト形式でプログラミング言語のような特化言語を作成する DSL である。内部 DSL は主にプログラム言語を拡張して作成されているためテキスト DSL が殆どである。テキスト形式であるため記述が容易で、エディタによるサポートがあるものも多い。また、テキストの検索や置き換えといったテキスト処理との親和性も高い。テキスト DSL を提供するツールとして Xtext[9][10] がある。

グラフィカル DSL は DSL を UML のような図形式で表現する DSL である。図形式であるため、要素間の関係を俯瞰することが容易であるが、記述要素が増えると可読性が低減する。グラフィカル DSL を提供するツールとして、MetaEdit[11], GMF[12], Graphiti[13], Sirius[14] がある。

2.7 組込みシステム向けモデリング言語

既存のハードウェアに関するモデルを扱えるモデリング技術やその体系として、MARTE[15] や SysML[16], AUTOSAR[17] などが挙げられる。SysML はシステムレベルの記述を行うことが出来る UML[18] の拡張・サブセット言語でハードウェアの論理構成を記述することが出来る。AUTOSAR は車載システムに特化した記述言語で組込みシステムのハードウェア、ソフトウェアを統合して扱うことが出来る。

MARTE は UML の組込みシステム向けプロファイルで、組込みシステムの設計や性能評価に用いられる。このためある程度詳細にハードウェアの記述を行うことができる。

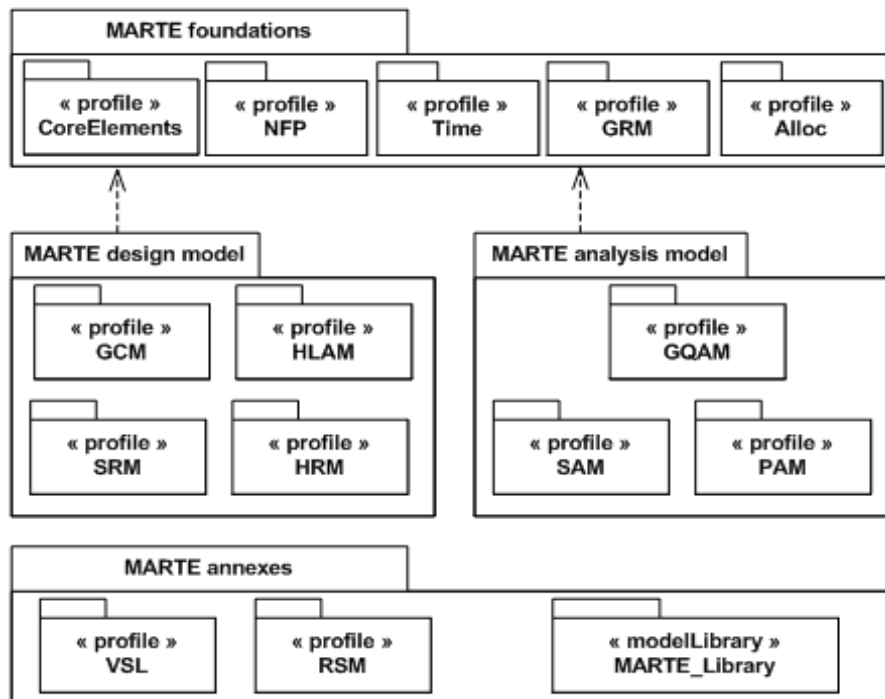


図 2.7: MARTE のパッケージ構造

MARTE のパッケージ構造を図 2.7 に示す¹。MARTE は大きく 4 つのパッケージに分類されるが、このうち MARTE foundations は MARTE 全体で用いる基盤となるモデル群を提供しており、MARTE annexes は組み込みシステムで用いられる型や単位などをモデル化したものである。

これらのパッケージの基盤として、ソフトウェアとハードウェア構造を記述する MARTE design model と組み込みソフトウェアの性能評価に関する情報を付与する MARTE analysis model が定義されている。

ここでは本研究に用いる MARTE design model について述べる。GCM(Generic Component Model) パッケージは MARTE design model で共通するモデルをまとめたもので他のパッケージから利用される。HLAM(High-Level Application Modeling) は組み込みシステムをサービスレベルで記述するためのモデルである。

SRM(Software Resource Model) は組み込みソフトウェアのモデル化のためのパッケージである。UML に加え、並行性やメモリの扱いなど組み込みで重点的に用いら

¹Architecture of the MARTE Profile (UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems v1.1 p25)

れる要素をモデル化している。今回対象とするゲートウェイドライバのようなソフトウェアはSRMのSW_Brokingパッケージにてモデル化されている。

HRM(Hardware Resource Model)は組み込みハードウェアのモデル化のためのパッケージである。HRMは、HRMの共通モデルを扱うHW_General、ハードウェアの論理構成を表すHW_Logical、ハードウェアの物理構成を表すHW_Physicalの3つのパッケージで構成される。

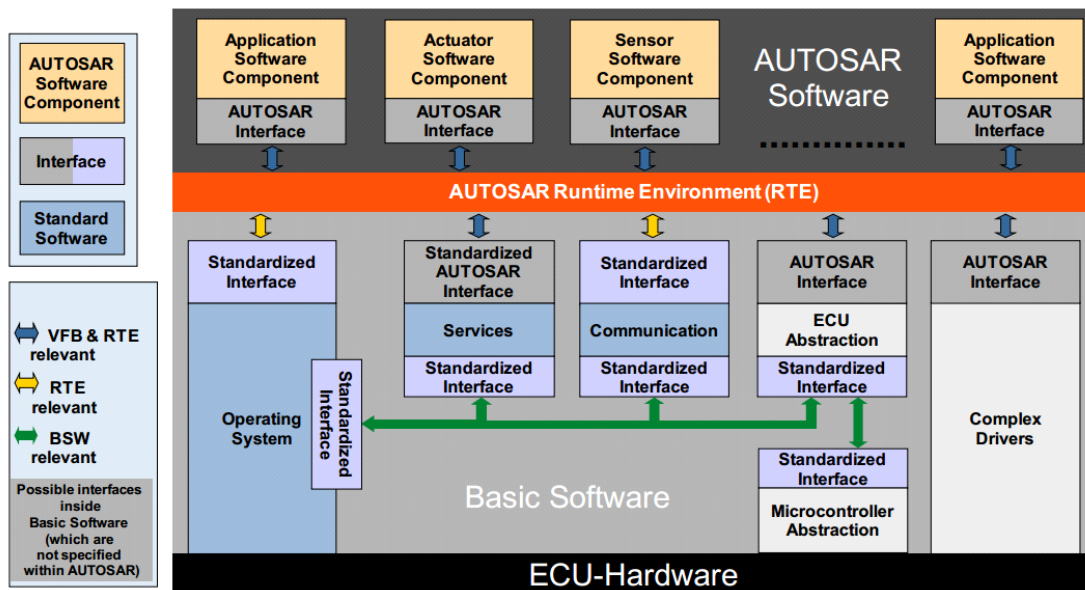


図 2.8: AUTOSAR 下位ソフトウェアレイヤ

一方、AUTOSARのパッケージ構造は図2.8にのようになっている²。AUTOSARでは、複数のECU(Electrical Control Unit:車載システムのMCU)を接続し動作させるシステムを想定しており、通信やハードウェアの差異を覆うAutosar RTE(RunTime Environment)で抽象化される。OSを含むデバイスドライバなどの下位ソフトウェアはBasic Softwareという形でモデリングを行う。

²Component View on the AUTOSAR layered software architecture AUTOSAR_EXP_VFB.pdf p32

第 3 章

問題

本章では、本研究の想定するシステムプロダクトライン開発とその実現における問題の分析を行う。

3.1 本研究の想定する組込みシステムプロダクトライン開発

問題分析に入る前に本研究で想定する組込みシステムプロダクトライン開発について述べる。

本研究では中小規模の組込みシステム、特にウェアラブル機器や IoT (Internet of Things) 機器、小型の家電などを想定する。このような組込みシステムは、ハードウェアプラットフォームの変化が大きく、機器ごとに異なったハードウェア構成を持つことが多い。また、このようなハードウェアプラットフォームでは、ハードウェアの仮想化はあまり行われておらず、プラットフォームに合わせたソフトウェアが必要となる。

こうした組込みシステムにおいても、製品系列を持ち類似した複数の製品を開発することが多い。また、同じハードウェアプラットフォームでもソフトウェアによって異なる製品としているものもある。

一般に組込みシステム開発では、ハードウェアとソフトウェアに切り分けて開発が行われる。ハードウェアの開発とソフトウェアの開発は並行して行われ、そ

れぞれが製品系列を持つ。ソフトウェアプロダクトライン開発ではソフトウェアの視点から製品系列を捉えるが、組込みシステムの開発の実態を考えると、従来からのソフトウェア部分にフォーカスしたプロダクトライン開発ではなく、ハードウェアの製品系列をふまえてソフトウェアの製品系列を整理するプロダクトライン開発が必要であると考えられる。

本研究は、このような中小規模でシステムプロダクトラインを持つ組込みシステムのソフトウェア開発を支援するものである。以降、このようなシステムプロダクトライン開発における問題点を取り上げる。

3.2 組込みシステムプロダクトライン開発における問題

上述したように組込みシステム開発をシステムプロダクトライン開発として捉える場合、ソフトウェアのプロダクトラインとハードウェアのプロダクトラインとの双方を考慮する必要があると考えられる。この場合各プロダクトはソフトウェアのプロダクトとハードウェアのプロダクトを組み合わせた構成となる。例えば製品1は、ハードウェア構成A、ソフトウェア構成a、製品2はハードウェア構成A、ソフトウェア構成b、製品3はハードウェア構成B、ソフトウェア構成cのように製品系列は、ハードウェアとソフトウェアの組み合わせからなる、というような構成となる。

また、ハードウェアの機能を利用するソフトウェアはハードウェアへの依存関係を持つ。このためハードウェアとソフトウェアの製品系列を考える際には、ソフトウェアの製品系列とそれが依存するハードウェア製品の間関係を適切に管理する必要がある。こうした依存関係が考慮されないまま、ハードウェアプロダクトとソフトウェアプロダクトが導出された場合、ハードウェアプロダクト上でソフトウェアプロダクトをテストする段階でソフトウェアプロダクトが動作しないといった不具合が発覚する可能性がある。このような事を防ぐため、ハードウェアとソフトウェアの依存関係を適切に定義するとともに、導出されたハードウェア製品で動作するソフトウェアフィーチャのみを選択できるようにすることが望ましい。

上記を踏まえて、システムプロダクトライン開発における問題として2つを取り

上げる。一つは、組込みシステムのハードウェアフィーチャとソフトウェアフィーチャの管理の問題である。組込みシステムの製品系列には、ハードウェアの製品構成とソフトウェアの製品構成があり、それぞれを適切に管理する必要がある。

もう一つは、ハードウェアフィーチャとソフトウェアフィーチャの依存関係の問題である。システムプロダクトラインではハードウェアフィーチャとソフトウェアフィーチャを扱うが、ハードウェアとソフトウェアのフィーチャ間には上述したような依存関係があるので、それを管理する必要がある。

3.3 ゲートウェイドライバ導出における問題

一般にプロダクトライン開発では、特定の製品のフィーチャを指定し、プロダクトを導出した際にそのプロダクトに対応する製品を体系だてて導出することが必要となる。システムプロダクトラインにおいてもシステム製品の導出が必要となるが、導出を実現するためには、ハードウェア製品の導出、ソフトウェア製品の導出だけでなく、両者のインタフェースとなるゲートウェイドライバの導出を体系的に行う必要がある。ゲートウェイドライバは、明示的な製品フィーチャとしては見えないため、導出はできるだけ自動化されることが望まれる。

ゲートウェイドライバはハードウェアの構成に依存するため、開発にはハードウェアの情報が必要となる。必要となるハードウェア情報として、ハードウェアの論理的な構成を表すブロック図、電気的な構成を表す回路図、個々の部品の詳細を示すデータシートの三つが利用されることが多い。ゲートウェイドライバに必要なブロック図、回路図、データシート情報はハードウェアフィーチャモデルから選択されたハードウェアプロダクトに基づくハードウェア設計から得られる。

開発者はこれらの情報の中からゲートウェイドライバ開発に必要な情報を選別するとともに、ハードウェア視点で記述された情報をソフトウェア視点から解釈し、それに基づいてゲートウェイドライバを開発しなければならない。特にデータシート情報は複数の観点の情報が含まれ、フォーマット等も多様であるため、これらの作業にはハードウェアの知識と実務的な熟練が必要である。

これらを踏まえ、ゲートウェイドライバの導出における問題として、ハードウェアの多様性、ハードウェア情報の多様性、開発手順の暗黙性の3つを取り上げる。

3.3.1 ハードウェアの多様性

一般にMCUやデバイスの選択肢は多岐に渡り、多くの製品がラインナップされている。特にシステムのプロダクトライン開発においては、様々なMCUやデバイスを組合せたハードウェア製品が導出されるため、様々なMCUやデバイスの組み合わせが作られる。

こうした組み合わせに基づいて作られるハードウェア製品に応じて、ソフトウェアとのインタフェースとなるゲートウェイドライバを提供する必要がある。開発を効率的に行うためには、こうしたゲートウェイドライバを体系だてて導出することが望まれるが、そのためには入力となるハードウェア情報を一貫した方法で形式化することが必要となる。

3.3.2 ハードウェア情報の多様性

図3.1にゲートウェイドライバ開発に必要なハードウェア情報について示す。ハードウェア情報は大きくハードウェア部品の情報と、ハードウェア構成の情報に分けられ、ハードウェア部品の情報にはMCUとデバイスのデータシート、ハードウェア構成の情報にはブロック図と回路図が含まれる。

これらの情報のうちハードウェア部品の情報である、MCUとデバイスのデータシートはメーカーや種類によってフォーマットが様々である。必要な情報がデータシートに含まれていても、フォーマットが多様であるため一律に機械処理することは難しい。またゲートウェイドライバ開発に必要なないハードウェア設計に関する情報や物理特性なども多く含まれる。このため、必要なハードウェア情報を取捨選択して適切に取り扱う方法が必要である。

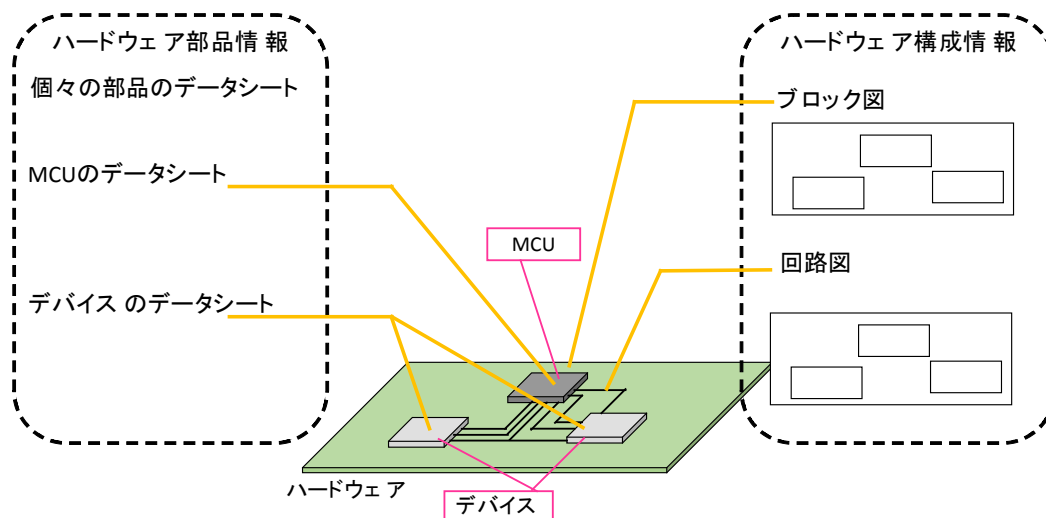


図 3.1: ハードウェア情報

3.3.3 開発手順の暗黙性

2.3.3 で取り上げたように、ゲートウェイドライバの開発方法は、開発者ごとに属人的な方法が取られることが多く、また明示的に定義されていないことが多かった。ゲートウェイドライバの開発にはハードウェアの知識や熟練が必要であり、また上述したようにデータシートなどが多様であるため、従来はハードウェアとソフトウェアの両方の知識を持った開発者が属人性の高い手順で暗黙知に基づいて開発を行うことが多い。ゲートウェイドライバ開発の体系だった導出のためには、開発手順を明示化して整理する必要がある。

また、ゲートウェイドライバの開発においては、ハードウェアの視点で定義された情報から、ソフトウェアから見た情報への視点変換が必要となる。ゲートウェイドライバの設計では、デバイスや周辺機能ごとにコンポーネント（クラス）を作成し、そのデバイスや周辺機能の持つ機能を API として上位アプリケーションに提供するとともに、それをどう制御するかという振る舞い設計を行い、API を実装する。ここでコンポーネントは、ハードウェアの論理構成であるブロック図より得られる。API はデバイスドライバではデバイスのデータシートの機能の項、ペリフェラルドライバでは MCU の周辺機能の項よりそれぞれの機能と制約を考

慮した上で作成する。この際、命名や引数の型の定義などの設計者の判断が必要となる。APIの振舞いの設計は、ブロック図、回路図、データシートのレジスタマップ、IOマップ、機能の制御方法など多くの項目を考慮した上で行う。こうしたハードウェア情報からソフトウェア情報への視点変換の手順化や支援が必要である。

第 4 章

アプローチ

本章では、前章であげた問題点を解決するためのアプローチを述べる。

4.1 全体像：組込みシステム向けモデル駆動システムプロダクトライン開発

3 章では、組込みシステムプロダクトライン開発の実現における問題点として、以下を指摘した。

- 組込みシステムプロダクトライン開発における問題
 - － ハードウェアとソフトウェアそれぞれのプロダクトライン
 - － ハードウェアフィーチャモデルとソフトウェアフィーチャモデルの依存関係
- ゲートウェイドライバ導出における問題点
 - － ハードウェアの多様性
 - － ハードウェア情報の多様性
 - － 開発手順の暗黙性

本研究では前者のシステムプロダクトラインの問題に対して、ハードウェアとソフトウェアのそれぞれのフィーチャモデルを作成し、それらの間に依存関係を定義した上で、段階的にフィーチャモデルを導出する手法を提案する。

また後者のゲートウェイドライバ導出の問題に対して、ハードウェア情報をモデルとして体系的に整理し、導出手順をモデル変換として定義することでモデル駆動技術による解決を提案する。

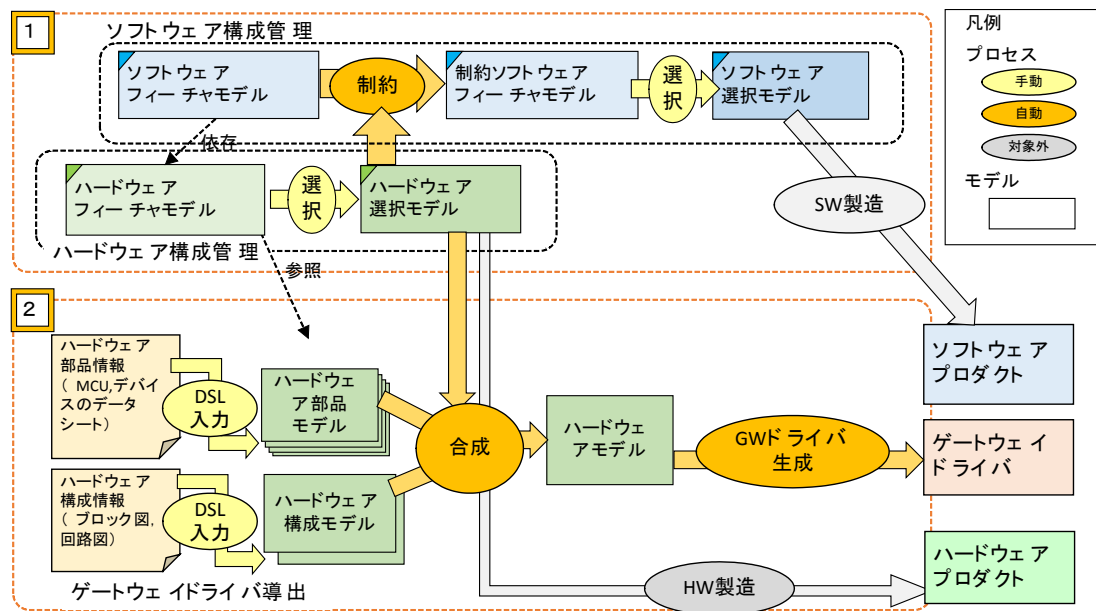


図 4.1: アプローチの全体像

図 4.1 に本研究のアプローチを示す。図中の 1 (以降, 1 は図中の橙色の二重四角の番号を指す) の囲いが段階的なフィーチャモデル導出によるシステムプロダクトラインのプロセス, 2 の囲いがモデル駆動技術によるゲートウェイドライバ導出のプロセスである。

1 と 2 のプロセスは関連しており, ハードウェアフィーチャモデルのハードウェアフィーチャは, 対応するハードウェア部品モデルに関連付けられている。また, ハードウェアフィーチャモデルから導出されるハードウェア選択モデルは, 1 のプロセスに入力され, ハードウェアプロダクトに対応するゲートウェイドライバの導出に用いられる。以後, 4.2 章にて 1 のプロセスを, 4.3 章にて 2 のプロセスの詳細を示す。

4.2 システムプロダクトラインによる構成管理

本章では、段階的なフィーチャモデルの導出によるシステムプロダクトライン開発について述べる。

4.2.1 段階的なフィーチャモデル導出のプロセス

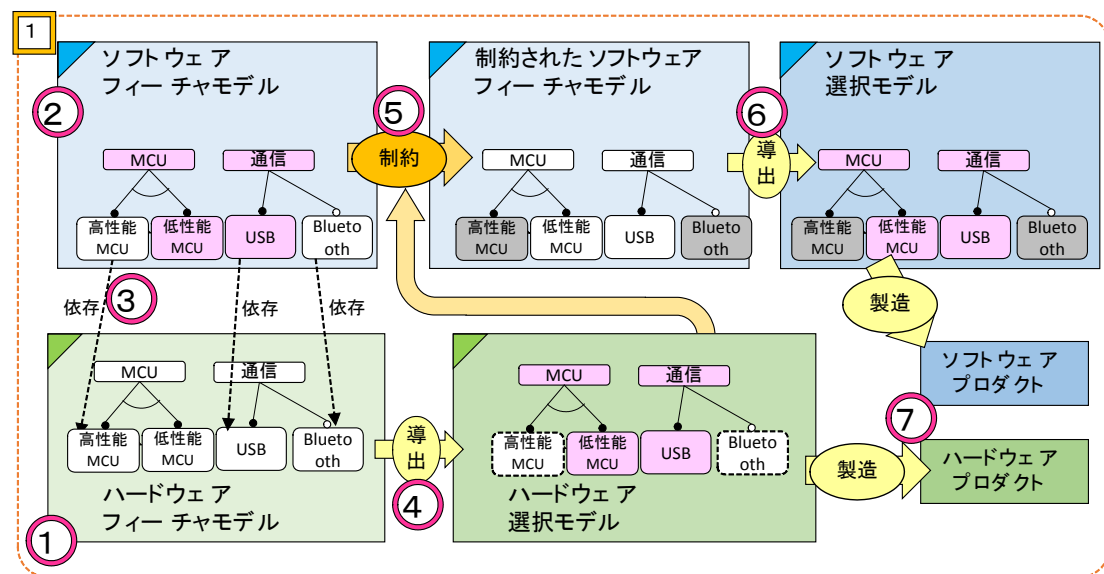


図 4.2: 段階的なフィーチャモデルの導出プロセス

図 4.2 に段階的なフィーチャモデル導出のプロセスを示す。本プロセスでは、ハードウェアフィーチャモデル（①，以降①は図中ピンク色の二重丸の番号を指す）とソフトウェアフィーチャモデル（②）の二種類のフィーチャモデルを用いる。加えて、ソフトウェアフィーチャモデルからハードウェアフィーチャモデルの依存関係を定義する（③）。

従来のシステムプロダクトライン開発でも多層のフィーチャモデルや複数のフィーチャモデルは用いられてきたが、層間の依存関係を考慮しながらフィーチャを選択するのは煩雑であるとともに、ハードウェアプロダクトの構成によっては、無効なソフトウェアプロダクトが導出されてしまう問題もあった。

本研究では、ハードウェアフィーチャモデルからハードウェア選択モデルを導出し (④), ハードウェア選択モデルに含まれる依存関係を辿り, 無効なソフトウェアフィーチャを排除した, 制約されたソフトウェアフィーチャモデルを導出する (⑤). 制約されたソフトウェアフィーチャモデルでは, 現在選択されているハードウェアプロダクトでは無効なソフトウェアフィーチャが除外され, 有効なソフトウェアフィーチャしか選択できない. このため, 制約されたソフトウェアフィーチャモデルではプロダクトの組み合わせ数が削減され, 整合したソフトウェアプロダクトを導出することができる. (⑥, ⑦)

4.2.2 ハードウェアフィーチャモデル

ハードウェアフィーチャモデルは, ハードウェア製品系列の共通性と可変性をフィーチャの観点から記述したモデルである. 一般にハードウェアはソフトウェアの機能やサービスを実現するためのプラットフォーム的な役割を果たすことが多いため, システム全体をひとつのフィーチャモデルで表現した場合の FORM[20] の Operational Environment Layer や Implementation Technique Layer に相当するフィーチャの多くがこのハードウェアフィーチャモデルに現れる. 本研究では, 一つのハードウェアフィーチャがひとつのハードウェアデバイスに対応する程度の記述粒度を想定している.

4.2.3 ソフトウェアフィーチャモデル

ソフトウェアフィーチャモデルは, ソフトウェアの製品系列の共通性と可変性をフィーチャの観点から記述したモデルである. 一般にシステムの外見の機能やサービスはソフトウェアで実現されるため, システム全体をひとつのフィーチャモデルで表現した場合の, FORM の Capability Layer に相当するフィーチャの多くがこのソフトウェアフィーチャモデルに現れる.

4.2.4 ハードウェアフィーチャとソフトウェアフィーチャの依存関係

ハードウェアを利用するソフトウェアフィーチャは、ハードウェアフィーチャに対して依存関係を持つ。ソフトウェアフィーチャからハードウェアフィーチャへの依存関係は、両モデル間に依存関係を表すモデルを付与することで定義する。依存関係はソフトウェアからハードウェアの一方向であるが、ハードウェアの構成からそれに基づくソフトウェアフィーチャを識別できるように、モデル上では双方向にモデルが辿れるように定義を行う。

ソフトウェアフィーチャモデルからソフトウェア製品のフィーチャを選択する際には、依存関係の一覧とハードウェア製品モデルで選択されたハードウェアフィーチャのセットを用いて、選択されていないハードウェアフィーチャへの依存関係を持つソフトウェアフィーチャを除外する形で制約を加える。

4.2.5 2つのフィーチャモデルの定義と利用

システムプロダクトラインを構成する際は、ハードウェアフィーチャモデルを作成する。この際ハードウェアフィーチャに対応するハードウェア部品モデルを関連付ける。続いて、ソフトウェアフィーチャモデルを作成する。ハードウェアに依存するソフトウェアフィーチャは対応するハードウェアフィーチャに対して依存関係を定義する。

製品導出の際は、まずハードウェアフィーチャモデルからハードウェア選択モデルを導出する。

続いてそのハードウェア選択モデル中のハードウェアフィーチャ以外に依存しているソフトウェアフィーチャを削除する。これにより、選択されたハードウェア製品で動作しないソフトウェアフィーチャは選べなくなる。

以上のように制約されたソフトウェアフィーチャモデルからはソフトウェア製品を導出する。以上の手順で、ハードウェア製品の構成情報とそのハードウェア上で動作するソフトウェアの構成情報の2つが得られる。

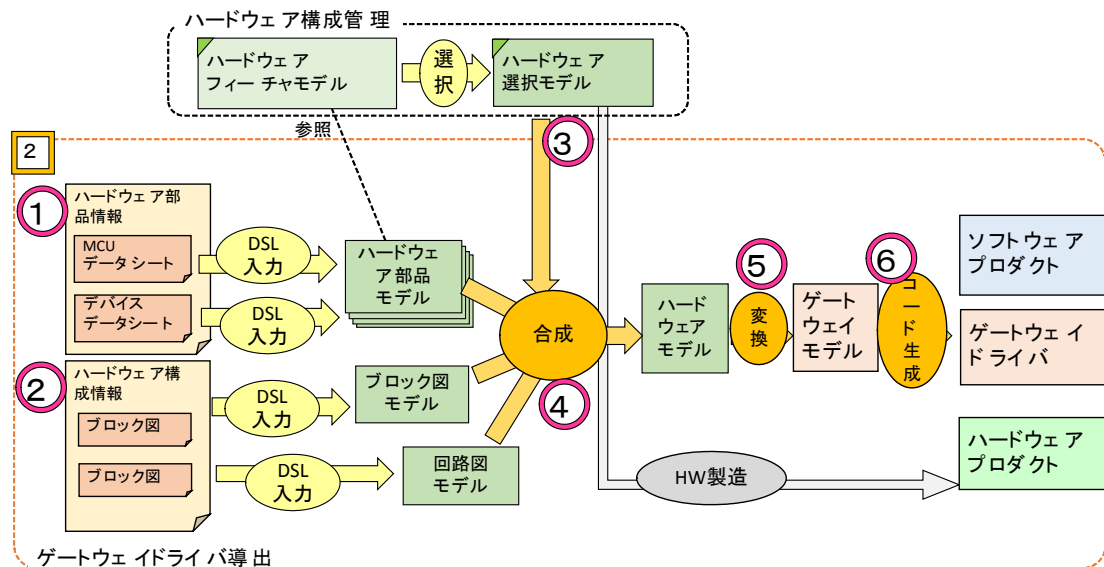


図 4.3: ゲートウェイドライバの導出

4.3 ゲートウェイドライバの導出

4.3.1 ゲートウェイドライバ導出のプロセス

図 4.3 にゲートウェイドライバ導出のプロセスを示す。ゲートウェイドライバの導出では、二種類のハードウェア部品情報（①）と二種類のハードウェア構成情報（②）とハードウェアフィーチャモデルから導出したハードウェア選択モデル（③）の計 5 種類のハードウェア情報を入力とする。

ハードウェア部品情報には MCU とデバイスのデータシートの情報が含まれ、データシートからは DSL を利用して情報を入力する。DSL はハードウェア部品モデルに変換される。またハードウェア部品モデルはハードウェアフィーチャモデルの対応するハードウェアフィーチャから参照される（①）。

ハードウェア構成情報にはブロック図、回路図の情報が含まれ、それぞれの形式に応じた DSL で入力する。DSL はブロック図モデル、回路図モデルに変換される（②）。

ハードウェア部品モデル、ブロック図モデル、回路図モデル、ハードウェア選択モデルは、ハードウェアモデルに変換される（④）。このハードウェアモデルは、ハードウェアフィーチャモデルから導出されたハードウェアプロダクトに

応じたゲートウェイドライバを導出するために必要となるハードウェア情報を全て含んだモデルとなる。

ハードウェアモデルはゲートウェイモデルに変換される (⑤)。ハードウェアモデルがハードウェア視点で構成されたモデルであるのに対し、ゲートウェイモデルはソフトウェア視点のモデルとなる。このため変換ではハードウェアからソフトウェアの視点変換を行っている。

ゲートウェイモデルからゲートウェイドライバのコード生成を行う (⑥)。ゲートウェイモデルには、ソフトウェアの構造モデルと振舞いモデルが含まれ、これらをコードテンプレートに入力することにより、ゲートウェイドライバのソースコードを出力する。

4.3.2 入出力モデルの形式化

前述したようにハードウェアの種類が豊富であるため、それらをすべて一貫した方法で形式化することは難しい。このため、対象となるハードウェアを限定し形式化を行う。また出力となるゲートウェイドライバも形式化する。

本研究で対象とするゲートウェイドライバでは、入力情報として、ハードウェア部品の情報とハードウェアの構成情報が必要となる。このため、ハードウェアの部品情報と構成情報を表現しうるハードウェアメタモデルを定義する。

また、出力であるゲートウェイドライバに対して、ゲートウェイモデルを定義する。ゲートウェイドライバは通常、特定の言語で実装されたソフトウェアとなるが、ソースコードとして生成するよりも、一旦モデルに変換した後にコード生成を行う形とする。一度モデルとすることにより、ゲートウェイドライバを利用するアプリケーションソフトウェアと設計モデル上で整合することができるとともに、特定の言語に非依存な再利用資産として利用することができる。

4.3.3 入力情報 DSL

定義したハードウェアモデルと、実際の入力となるハードウェア情報（データシートやブロック図）は表現方法が異なる。モデルの表現形式は一般に XML などのコンピュータで解釈するのに適した形式が用いられるが、このような形式の情

報の入力は人間が行うのには適さない。加えて入力情報のフォーマットはメーカー間で統一されておらず、自動化に不要な情報も多く含まれる。

このため、モデルの入力には DSL を用いて行う。入力情報である 3 つのハードウェア情報（ハードウェア部品のデータシート、ブロック図、回路図）のフォーマットに合わせ、3 つの DSL を定義する。データシートの DSL はテキスト形式、ブロック図の DSL はグラフィカル形式とする。回路図の DSL は多くの回路図エディタでエクスポートをサポートするネットリストを入力とするテキスト形式とする。

4.3.4 ハードウェアモデルからゲートウェイモデルまでの変換定義

形式化されたハードウェアモデルからゲートウェイモデルへの変換を定義する。変換においては、ペリフェラルドライバの導出、デバイスドライバの導出、初期化処理の導出、が必要となる。変換されたゲートウェイドライバはさらにコードへ変換される。

4.3.4.1 ペリフェラルドライバの導出

ペリフェラルドライバは、MCU の周辺機能の機能を提供するソフトウェアである。MCU によってはライブラリとして提供されている場合もある。周辺機能は多くの場合、制御用のレジスタがメモリにマッピングされており、MCU からはレジスタを介して周辺機能の制御を行う。周辺機能の制御手順はレジスタの操作手順で記述されており、機能ごとにレジスタの操作列として実装する。

4.3.4.2 デバイスドライバの導出

デバイスドライバの導出のためには、ハードウェア情報のうちブロック図、デバイスのデータシート（デバイス機能、デバイスの利用方法）、MCU のデータシート（周辺機能）が必要となる。まず、ブロック図より MCU とデバイスがどのように接続され、どのプロトコルが利用されているかを確認する。通常、デバイスドライバのデータシートのデバイスの利用方法は接続プロトコルの抽象度で記述さ

れている。例えば I2C を使っている場合であれば、I2C API レベルで手順が記述される。

多くの場合、接続プロトコルは MCU の周辺機能として実装されており、MCU から利用可能である。接続プロトコルの API は統一したものが存在する訳ではなく、MCU ごとに API が異なる場合や実装されていない場合がほとんどである。このため MCU の周辺機能の API（ペリフェラルドライバ）として読み替えて実装する。

4.3.4.3 初期化処理の導出

ゲートウェイドライバを利用するには、その初期化作業が必要である。そこでデバイスドライバ、ペリフェラルドライバの両方に対して、初期化処理の導出を行う。MCU には複数の周辺機能が搭載されており、どの周辺機能を利用するかを設定しなくてはならない。また周辺機能は MCU 外部と接続するため、MCU の IO と接続されている。このため IO を適切に設定する必要がある。

初期化の手順も周辺機能の利用方法の一部としてレジスタの操作手順が記載されており、実装は MCU の初期化ルーチン内にレジスタ操作列を追記して行う。

4.3.4.4 ゲートウェイドライバのコード生成

ゲートウェイモデルはモデルであるため、対象環境で動作するソースコードとしてゲートウェイドライバを生成する必要がある。ゲートウェイモデルは言語非依存であるため実装対象の言語に合わせて、テンプレートを用意しゲートウェイモデルを当てはめることで、ゲートウェイドライバのソースコードを生成する。

第 5 章

提案手法

本章では 4 章で説明したアプローチの具体的な実現方法について述べる。

5.1 システムプロダクトライン

4 章のアプローチで述べた通り，本研究では組込みシステムの製品系列の可変性分析をハードウェア，ソフトウェアの 2 つに分割し，それぞれフィーチャモデルを作成する．本研究ではこの 2 つのフィーチャモデルを合わせて 2 層フィーチャモデルと呼ぶ。

5.1.1 2 層フィーチャモデルのメタモデル定義

2 層フィーチャモデルのメタモデルは，ハードウェアフィーチャパッケージとソフトウェアフィーチャパッケージ，両パッケージの共通部分であるコアパッケージの 3 つのパッケージで構成される．コアパッケージではフィーチャモデルの基本的なモデル定義がなされる．ハードウェアフィーチャ要素とソフトウェアフィーチャ要素はコアパッケージのフィーチャ要素を継承し，ハードウェアフィーチャ，ソフトウェアフィーチャそれぞれの拡張して定義される。

ハードウェアフィーチャパッケージでは，コアパッケージのフィーチャ要素を継承したハードウェアフィーチャを定義し，ハードウェアフィーチャは後述する

ハードウェアモデルを参照する。ソフトウェアフィーチャパッケージでは、ハードウェアフィーチャ要素への依存関係を表す制約要素が定義されている。

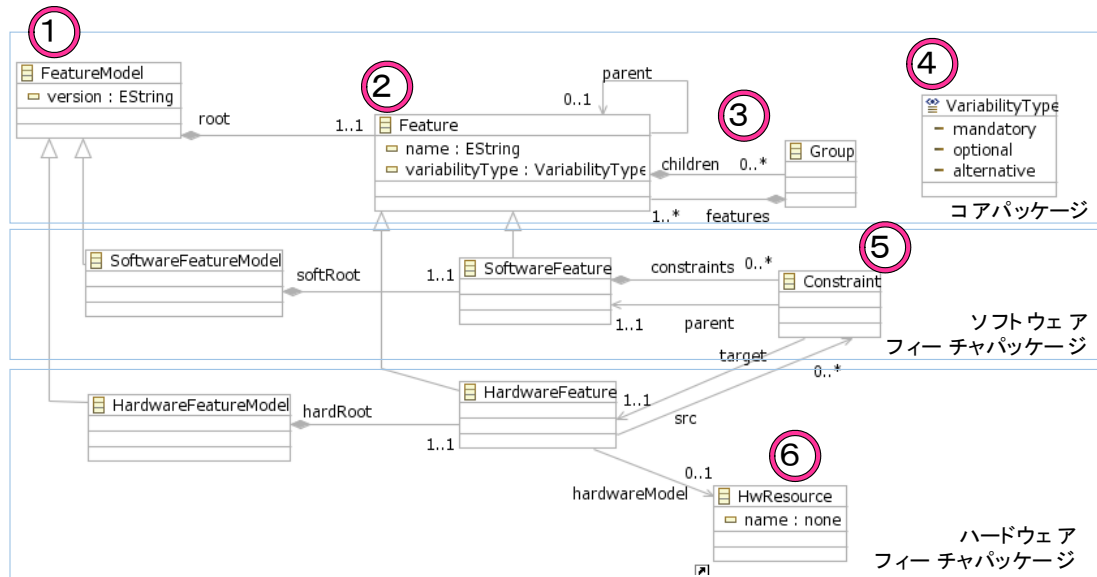


図 5.1: 2層フィーチャモデルのメタモデル

図 5.1 に 2 層フィーチャモデルのメタモデルを示す。

- ① :FeatureModel : フィーチャモデル自体を表すモデル，子要素として，ハードウェアフィーチャモデル，ソフトウェアフィーチャモデルを持つ。また，ルートとなるフィーチャ要素への関連を持つ。
- ② :Feature : フィーチャを表すモデル。属性としてフィーチャ名とフィーチャのタイプ（④ 参照）を持つ
- ③ :parent, children : フィーチャは木構造となっており，単一のルート要素を根として複数の子となるフィーチャを持つ。複数の子要素は Group として纏められている。
- ④ :VariabilityType : フィーチャのタイプ。Mandatory（必須），Optional（選択），Alternative（排他）の列挙型
- ⑤ :Constraint : フィーチャ間制約，ここではソフトウェアフィーチャからハードウェアフィーチャへの依存関係を表す。

- ⑥ :HardwareModel : ハードウェアフィーチャはハードウェア部品モデルへの関連を持つ

5.1.2 2層フィーチャモデルの利用

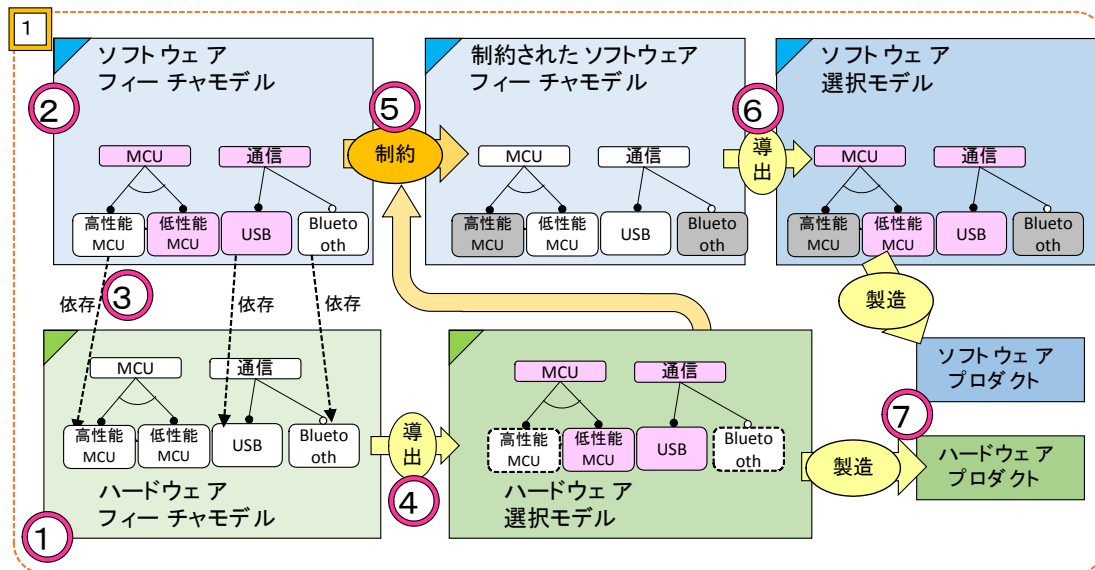


図 5.2: 2層フィーチャモデルを用いた製品導出のプロセス

図 5.2 に 4.2.1 で示した 2 層フィーチャモデルを用いた製品導出のプロセスを再掲する。

導出されたハードウェア選択モデルから，制約されたソフトウェアフィーチャモデルの導出方法について，具体的に説明する。

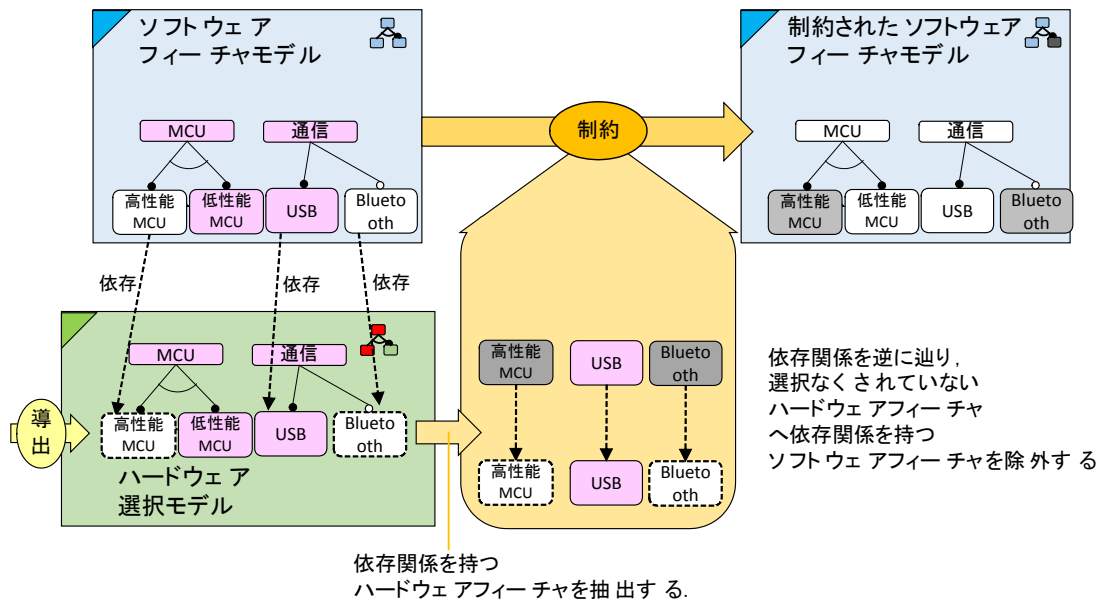


図 5.3: 制約されたソフトウェアフィーチャモデルの導出

図 5.3 にハードウェア選択モデルと依存関係から制約されたソフトウェアフィーチャモデルを導出するプロセスを示す。

まず、ハードウェアフィーチャモデルのすべてのハードウェアフィーチャのうちソフトウェアフィーチャからの依存関係を持つハードウェアフィーチャを抽出する。次にハードウェア選択モデルで選ばれていないハードウェアフィーチャの依存関係を逆に辿り、依存関係を持つソフトウェアフィーチャを除外する。

この際、ソフトウェアフィーチャが子要素を持つ場合は、子要素もすべて除外する。

5.2 ゲートウェイドライバ導出

5.2.1 ゲートウェイドライバの導出プロセス

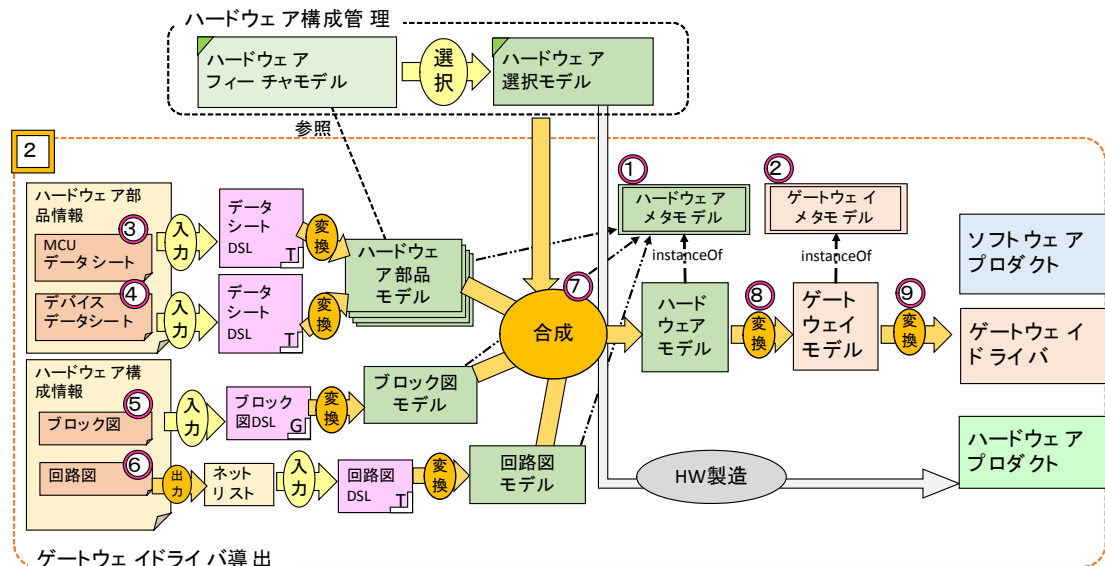


図 5.4: ゲートウェイドライバの導出プロセス

図 5.4 にゲートウェイドライバの導出プロセスの詳細図を示す。図内の小さな○数字に沿って説明する。

①，②：本手法を適用するために必要なメタモデルをあらかじめ定義しておく。ハードウェアメタモデルを定義するために、どのようなハードウェア情報が必要か分析を行ったが、それについては 5.2.2 に説明する。定義したメタモデルについては 5.2.3 で説明する。

③，④，⑤，⑥：ゲートウェイドライバ導出に必要なハードウェアの情報を DSL を使って入力する。データシート等の情報と、ハードウェアモデルでは表現形式に乖離があるので、まずデータシート等での表現に近い形式を持った DSL を使って入力を行い、それを変換してハードウェアモデルを得る。DSL についての詳細は 5.2.4 で説明する。

⑦：DSL から変換された複数のモデルとハードウェア選択モデルを用いて、そのハードウェア選択モデルに対応したゲートウェイドライバの導出に必要なハー

ドウェア情報をすべて含んだハードウェアモデルを合成する。合成方法については5.2.5で説明する。

⑧:合成されたハードウェアモデルを変換してゲートウェイモデルを得る。変換方法については5.2.6で説明する。

⑨:変換されたゲートウェイモデルからコード生成を行い、ゲートウェイドライバを生成する。コード生成についても5.2.6で説明する。

5.2.2 ハードウェア情報の選択

本研究では、入力するハードウェア情報として、MCUとデバイスのデータシート、ブロック図、回路図を想定している。4.3.2で述べたように、ハードウェアは多様であるため対象をある程度限定することによって形式化を行う。典型的なハードウェアを対象にそのデータシートの分析を行い、ハードウェアモデルのメタモデルを決定する。

5.2.2.1 MCU データシートの情報

以下に対象とするハードウェアの典型例としてAtmel社のMCUであるAtmega328のデータシートのアウトラインを示す。このデータシートは典型的な内容を含んでいるのでこれを例に分析内容を説明する。

行頭に*がついた項目は、ハードウェア情報のうちソフトウェア開発に関する情報で、ハードウェアモデルとして扱う情報である。また-がついた項目はソフトウェア開発に関する情報であるが、ハードウェアモデルから除外した情報である。印のない項目は、ソフトウェア開発に関係しないため除外する。

- 0.Features
- * 1.Pin Configuration
- 2.Overview
- 3.Resources
- 4.Data retention
- 5.About Code Examples
- 6.Capacitive Touch Sensing
- 7.AVR CPU Core
- * 8.AVR Memories

- 9.System Clock and Clock Options
- 10.Power Managements and Sleep Modes
- 11.System Control and Reset
- 12.Interrupts
- 13.External Interrupts
- * 14.I/O Ports
- 15.8-Bit Timer/Counter0 with PWM
- 16.16-bit Timer/Counter1 with PWM
- 17.Timer/Counter0 and Timer/Counter1 Prescalers
- 18.8-bit Timer/Counter2 with PWM and Asynchronous Operation
- * 19.SPI - Serial Peripheral Interface
- * 20.USART0
- * 21.USART in SPI Mode
- * 22.2-wire Serial Interface
- 23.Analog Comparator
- 24.Analog to Digital Convertor
- 25.debugWire On-chip Debug System
- 26.Self-Programming the Flash, Atmega 48A/48PA
- 27.Boot Loader Support - Read-While-Write Self-Programming
- 28.Memory Programming
- 29.Electrical Characteristics - (TA=-40 °C to 85 °C)
- 30.Electrical Characteristics - (TA=-40 °C to 105 °C)
- 31.Typical Characteristics - (TA=-40 °C to 85 °C)
- 32.Atmega48PA Typical Characteristics(TA=-40 °C to 105 °C)
- 33.Atmega88PA Typical Characteristics(TA=-40 °C to 105 °C)
- 34.Atmega168PA Typical Characteristics(TA=-40 °C to 105 °C)
- 35.Atmega328PA Typical Characteristics(TA=-40 °C to 105 °C)
- * 36.Register Summary
- 37.Instruction Set Summary
- 38.Ordering Information
- 39.Packaging Information
- 40.Errata
- 41.Datasheet Revision History

以下に各項目の概要を述べるとともに、除外するものについてはその理由を説明する。まず、選択した要素（1.8.14.19-22）について述べる。1. は MCU のピン

構成について示されている。ピンは MCU とデバイスの電氣的な接続関係を扱う情報となる。8. は MCU のメモリ構成について記述されている。一般に MCU のメモリ空間はプログラム領域，RAM 領域，レジスタ領域からなる。レジスタ領域は MCU の様々な設定や周辺機能进行操作するためのレジスタがマッピングされている。また，36. はメモリに割当てられているレジスタの一覧が記載されている。8 の情報を一覧で示したものである。

14. は I/O の詳細が記述されている。I/O の諸元に加え，割当てられている周辺機能やその I/O を利用するためのレジスタの設定方法などが記載されている。15-28 は各周辺機能についての詳細が記載されている，このうち代表的な通信に関する周辺機能である SPI(19)，USART(20,21)，I2C(22) についてモデル化を行う。

次にソフトウェア開発に関連するが今回は除外したもの（9-13,15-18,23-28,37）について述べる 9 は MCU のクロックに関する情報が含まれる。標準的な利用法であればクロック設定の必要はないが，クロックの倍率の変更や利用するクロックハードウェアによって MCU の初期化処理が必要な場合もある。今回は標準的な使い方を以るものとして，除外する。10 は電源管理やスリープに関する情報が含まれる。多くの MCU では省電力化のために MCU の動作を一時停止させる Sleep モードが搭載されている。今回は後述する割り込み機能を除外したため，割り込み処理の併用が必要なスリープ処理も除外する。11 はリセットに関する情報が含まれる。MCU には複数のリセット処理が用意されており，電源投入時のリセット，RESET ピンによるリセット，プログラムの暴走検知のためのウォッチドックタイマ（Watch Dog Timer：WDT），電圧低下時のブラウンアウトリセット（Brown Out Reset）などがある。ソフトウェア開発に深く関連するものとして WDT が挙げられるが，WDT も割り込みによる処理が必要となるため，除外する。12,13 は割り込み処理に関する情報が含まれる。割り込み処理はソフトウェア開発でも重要な要素であるが，扱うソフトウェア構造が複雑になるため，除外している。

25-28 は，デバッグ機能についてや，自己プログラミング，メモリプログラミングに関して記述されている。主に MCU 内部で用いる機能であるため，除外とした。37 は MCU の CPU の命令セットの一覧である。アセンブラによるプログラムは対象としないため，除外した。

その他の項目は，ハードウェアの電氣的特性や，物理的特性，発注方法とデー

タシートの版や正誤表などに関する情報であるため除外した。

5.2.2.2 デバイスデータシートの情報

デバイスについても同様に、データシートを分析してハードウェアメタモデルに含める情報を決定した。以下にデバイスの典型例として MPU9150 のデータシートを用いて説明する。なお*や-の印については 5.2.2.1 と同様である。

- 1 REVISION HISTORY
- 2 PURPOSE AND SCOPE
- 3 PRODUCT OVERVIEW
- 4 APPLICATIONS
- * 5 FEATURES
- 6 ELECTRICAL CHARACTERISTICS
- * 7 APPLICATIONS INFORMATION
- 8 PROGRAMMABLE INTERRUPTS
- * 9 DIGITAL INTERFACE
- 10 SERIAL INTERFACE CONSIDERATIONS
- * 11 ASSEMBLY

別紙（レジスタマップ）

- 1 REVISION HISTORY
- 2 PURPOSE AND SCOPE
- * 3 REGISTER MAP FOR GYROSCOPE AND ACCELEROMETER
- * 4 REGISTER DESCRIPTIONS FOR GYROSCOPE AND ACCELEROMETER
- * 5 REGISTER MAP FOR MAGNETOMETER
- * 6 REGISTER DETAILED DESCRIPTIONS FOR MAGNETOMETER

5.2.2.3 データシート情報の選別

MCU とデバイスのデータシートのアウトラインの一例を示した。これらのデータシートは一例であるが、MCU やデバイスのデータシートであればおおよそ同様の項目が含まれる。

このため、メタモデルを定義するにあたり扱う情報を内容に応じてカテゴリ化した。MCU に関しては以下の 4 つのカテゴリに分けた。

- IO 情報
- メモリ・レジスタ情報
- 周辺機能
- パッケージ情報

一方，デバイスに関しては以下のカテゴリに分けた．

- IO 情報
- メモリ・レジスタ情報
- デバイス機能
- パッケージ情報

周辺機能とデバイス機能の項目以外は，ほぼ共通して扱えることが分かる．

5.2.3 ハードウェア情報とゲートウェイドライバの形式化

前項にて，データシート情報の整理を行った．MCUとデバイスのデータシート項目を結合したものととして下記が挙げられる．

- IO 情報
- メモリ・レジスタ情報
- 周辺機能（MCUのみ）
- デバイス機能（デバイスのみ）
- パッケージ情報

これらの項目をハードウェアメタモデルとして形式化した．

5.2.3.1 MARTE の拡張

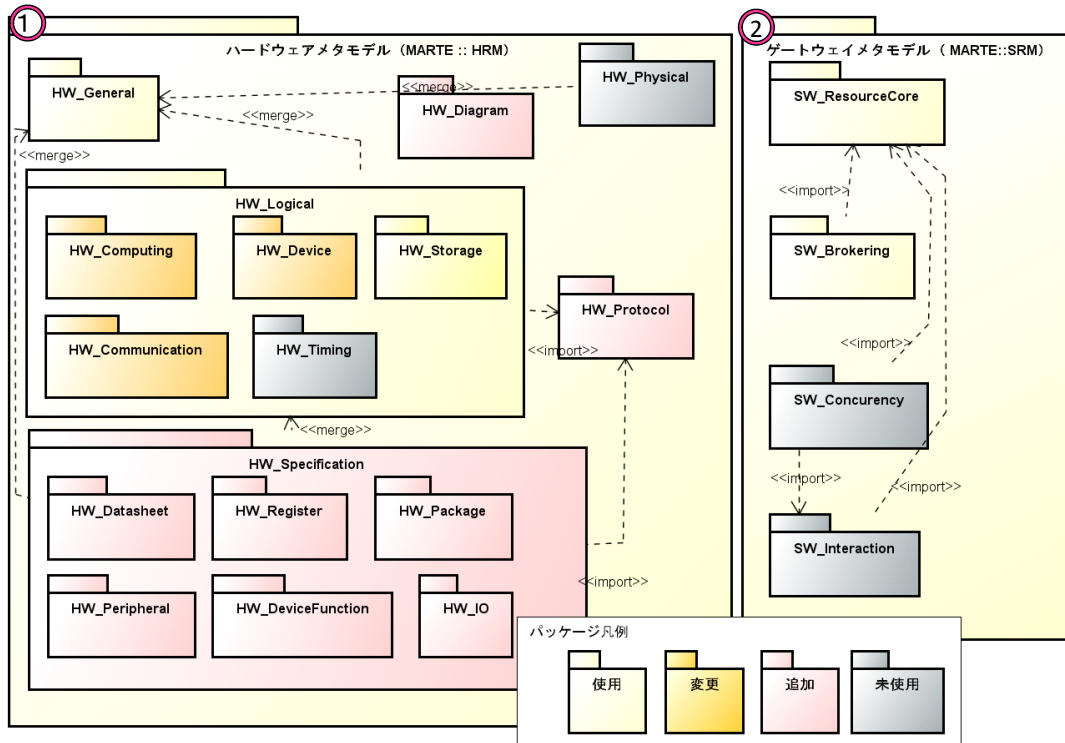


図 5.5: 拡張した MARTE のパッケージ構造

本研究で用いるメタモデルのパッケージ構造を図 5.5 に示す。

本研究ではハードウェア情報をモデル化するため、ハードウェア自体を表すモデル記述が必要となる。MARTE は組込みシステム向けの UML プロファイルで、既存のハードウェア記述モデルの中では、本研究で求めるハードウェアのモデル化に一番近いので、ハードウェアモデルの表現に MARTE を利用する。しかしながら、MARTE では上述した必要な項目をすべて表現することができないため、それらの項目を表現できるように拡張を加えた。

メタモデルの定義には、MARTE のパッケージのひとつである MARTE design model5.5 パッケージに含まれる Hardware Resource Model(HRM) パッケージと Software Resource Model (SRM) パッケージを利用する。HRM はハードウェアモデルの基盤として、SRM はゲートウェイモデルの基盤として利用する。

図内のピンク色の要素が新たに加えたパッケージである。主に HW_Specification

パッケージが新たに追加したものである。

5.2.3.2 メタモデルと各図の対応

以下にメタモデルのパッケージとゲートウェイドライバ導出のプロセスで用いる各モデル図の対応を示す。

- ハードウェア部品モデル - ハードウェアメタモデル (Hw_Specification)
- 回路図モデル - ハードウェアメタモデル (Hw_IO, Hw_Package)
- ブロック図モデル - ハードウェアメタモデル (Hw_Communication)
- ハードウェアモデル - ハードウェアメタモデル (Hw_Specification)
- ゲートウェイモデル - ゲートウェイメタモデル (SW_Brokering パッケージ)

5.2.3.3 ハードウェアメタモデル

本研究ではハードウェアモデルとして、ハードウェアの論理的な構成、電気的な構成、ハードウェア部品の詳細定義などが必要となる。HRMでは、ハードウェアのコンポーネントレベルを識別できるがコンポーネント内部の詳細なハードウェアは記述できない。このため、HRM への拡張を行った（図 5.5 内 Hw_Specification パッケージ）。

表 5.1 に主要なハードウェアメタモデルの要素を示す。

5.2.3.4 ゲートウェイメタモデル

ゲートウェイメタモデルは MARTE の SRM を基盤として拡張を行う。MARET SRM は、ソフトウェアの記述を行うためのモデルであるが、ゲートウェイドライバのようなハードウェア制御のための振舞いの定義は行えない。

このため、上記ハードウェアモデルで定義したレジスタなどのハードウェアとソフトウェアの接点となる要素を参照し、これに対する操作を行う要素を追加した。

| パッケージ | メタモデル名 | 概要 |
|-----------------|----------------|-----------------|
| HwComputing | HwMCU | MCU |
| HwDevice | HwDevice | Device |
| HwGeneral | HwResource | ハードウェア部品の抽象クラス |
| HwCommunication | HwPort | MCU・デバイスのポート |
| HwCommunication | HwCo | MCU・デバイス間の論理接続 |
| HwIO | HwPin | MCU・デバイスのピン |
| HwIO | HwLine | MCU・デバイスの電気接続 |
| HwPackage | HwPackage | 封入パッケージ |
| HwPackage | HwPackagePin | パッケージのピン |
| HwPackage | HwWire | パッケージ間の接続 |
| HwProtocol | HwProtocol | プロトコル |
| HwPeripheral | HwPeripheral | 周辺機能 |
| HwPeripheral | OperationImpl | 周辺機能のプロトコル実装 |
| HwPeripheral | RegisterAction | レジスタ操作アクションの抽象 |
| HwDiagram | *Diagram | 各図のコンテナとなるメタモデル |

表 5.1: 主要なメタモデル要素

5.2.4 ハードウェア情報入力用 DSL

ハードウェアモデルにはブロック図，回路図，MCU/デバイスのデータシートより多くの情報の入力が必要となる．このため各入力情報に応じて DSL を作成して，入力の支援を行う．

DSL は3種類定義する．ハードウェア部品 DSL では MCU とデバイス両方の記述ができる DSL を定義する．回路図 DSL は回路図エディタからエクスポートできるネットリスト形式を DSL として定義する．ブロック図 DSL はグラフィカル DSL で，図形式でブロック図を入力できる DSL を定義する．

5.2.4.1 ハードウェア部品 DSL

データシートの入力は，データシートに含まれる情報のうち，共通項を分析した上で本提案に必要な部分を入力するテキスト DSL として実装する．入力した DSL はハードウェアモデルに変換され後段のモデル変換へと渡される．

以下にハードウェア部品 DSL (MCU, Device 共通部) の定義の一部抜粋を示す．

Code 5.1: ハードウェア部品 DSL の定義 (Xtext 形式, 一部抜粋)

```
grammar jp.ac.jaist.kslab.sb.marte.spec.SpecDsl with
org.eclipse.xtext.common.Terminals generate specDsl "http://www.ac
    .jp/jaist/kslab/sb/marte/spec/SpecDsl"
import "http://www.eclipse.org/emf/2002/Ecore"
Model:
    Datasheet|DslProtocol;

Datasheet :
    ('revision' revision=PackageName)?
    'datasheet' name=ID
    '{'
        (imports += Import)*
        (components+=DslComponent)*
    '}' ;

Import:
    'import' importedNamespace = QualifiedNameWithWildcard;

DslProtocol:
```

```

        'protocol' name=QualifiedName '{'
            operations+=DslOperation*
        }';

DslOperation:
    (type=DslType)? name=ID '('(paramType+=DslType paramname+=
        ID (','
        paramType+=DslType paramname+=ID)*)?')';

DslComponent :
    DslMcu|DslDevice;

DslMcu :
    'mcu' name=ID '{'
        'pins_{' pins+=DslPin (',' pins+=DslPin)* '}'
        ports+=DslPort*
        peripherals+=DslPeripheral*
        ('sfrs_{'
            sfr+=DslRegister (',' sfr+=DslRegister)*
        }')?
        packages+=DslPackage*
    }';

DslDevice:
    'device' name=ID '{'
        'pins_{' pins+=DslPin (',' pins+=DslPin)* '}'
        ports+=DslPort*
        functions+=DslFunction*
        packages+=DslPackage*
        ('registers' '{'
            registers+=DslRegister (',' registers+=DslRegister
                )*
        }')?
    }';

```

③ ④

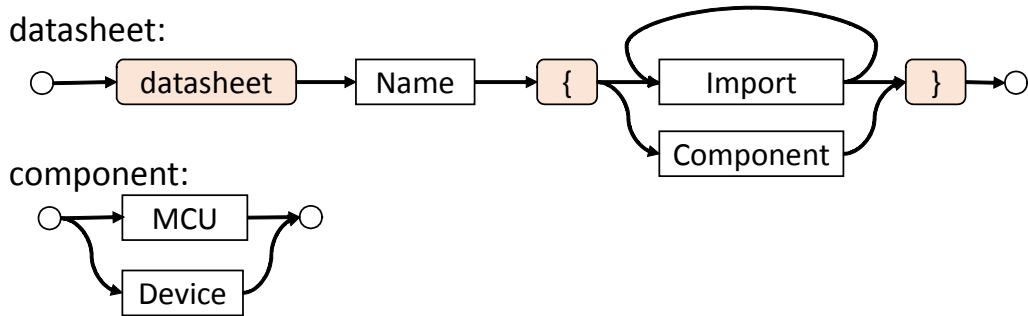


図 5.6: ハードウェア部品 DSL の定義

Code5.1 をグラフ化したものを図 5.6 に示す。(図形式では簡略化のため接頭語 Dsl を除去している。) 図では記載していないが、ルートの要素として Model を定義している。Model では datasheet または Protocol である。Protocol は MCU とデバイス間の接続プロトコルのインタフェース定義を行うためのもので、プロトコル名とそのプロトコルの持つメソッドの列挙を定義する。

定義 datasheet では、データシート自体を示す定義を行い、内部に Import または Component を含む。Import 文は別ファイルで定義した Protocol を参照するために用いる。Component は、MCU または Device で定義される。これにより、MCU とデバイスの両方のデータシートを扱うようにしている。

③

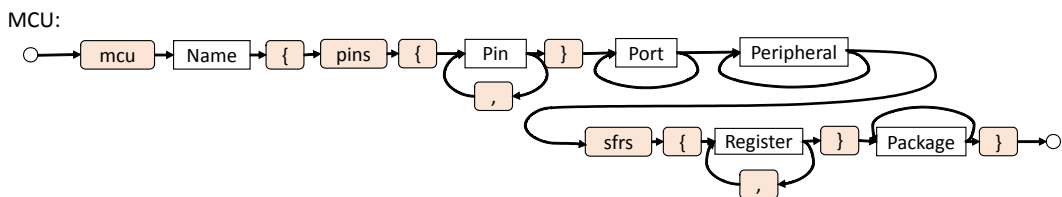


図 5.7: ハードウェア部品 DSL (MCU) の定義 (一部抜粋)

図 5.6 より、MCU をより掘り下げた DSL 定義を図 5.7 に示す。5.2.3 で示したように MCU データシートの項目として IO, メモリ・レジスタ, 周辺機能, パッケージを取り上げる。DSL では、それぞれ Pin(IO), Peripheral (周辺機能), Register

(メモリ・レジスタ), Package (パッケージ) の定義を行っている。また Port は周辺機能が用いる Pin のセットを示すためのものである。

Pin の定義は, MCU の論理的なピン名の列挙を行う。一方 Package では, パッケージのピン名とピン番号と割当てられている論理ピン名を記述する。例えば, Atmega328 の MLF パッケージでは, 1 番ピンのピン名が PD3, 割当てとして PCINT19, OC2B, INT1 が指定されている。この場合, Pin には PD3, PCINT19, OC2B, INT1 を列挙し, PackagePin には, ピン名として PD3, ピン番号 1, 他割り当てとして PCINT19, OC2B, INT1 を記述する (表記は PD3(1){PCINT19,OC2B,INT1} となる)。

Port は Port 名と, その Port に属する Pin を列挙する。Register はレジスタ名と割当てられているアドレスを記述する。

Peripheral の表記は多少複雑である。Peripheral として周辺機能名を記述し, その周辺機能の持つ機能を Operation として複数記述していく。Peripheral の定義ではその周辺機能の提供するプロトコルを指定している。関係としては Java で例えるとプロトコルがインタフェースで Peripheral がそれを実装するクラス, Operation がインタフェースの持つメソッドを実装したものとなる。

Operation 内には, その機能を利用するための操作手順を, レジスタの操作として記述する。操作手順はプログラム言語のような表記を行うため, レジスタ操作の記法に加えて簡易な算術演算や制御構文を記述できるようにしている。このような拡張は Xtext の言語拡張ライブラリである Xbase を用いて行っている。

③

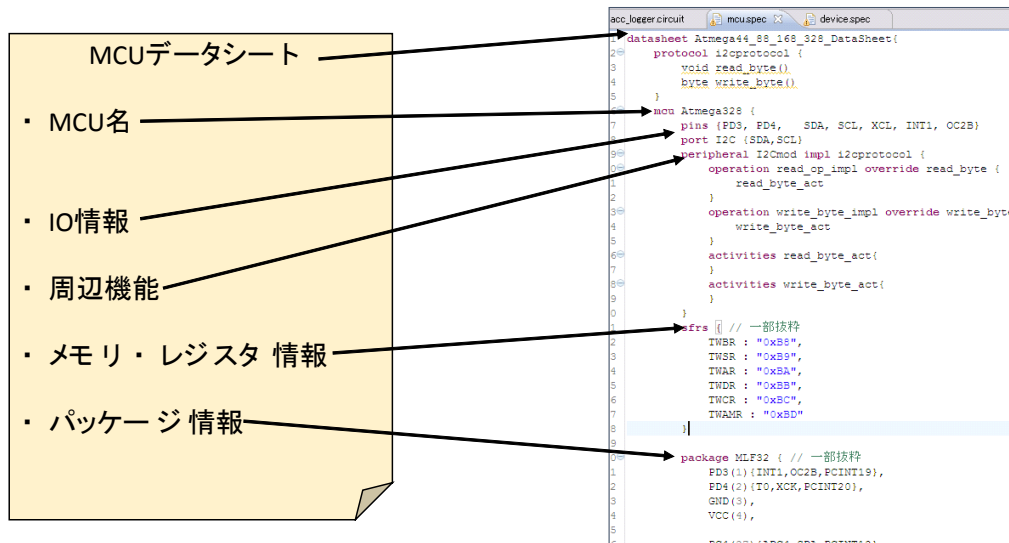


図 5.8: ハードウェア部品 DSL (MCU) の入力例

図 5.6 より, Device をより掘り下げた DSL 定義を図 5.7 に示す. MCU とデバイスは同じ DSL 記法で記すが, 入力項目は若干異なる. Pin, Port, Register, Package は MCU と同様の表記を行う. Peripheral は, デバイスではソフトウェアからは明示的に利用することはないため除外している.

DeviceFunction ではデバイス機能の定義を行う. 粒度としてはメソッドの定義にあたる. 内部の表記法は上記の Peripheral の Operation と同じものであるが, Operation がレジスタの操作を記述していたのに対して, DeviceFunction では Protocol で定義されているメソッド呼び出しの形で操作を記述する.

③

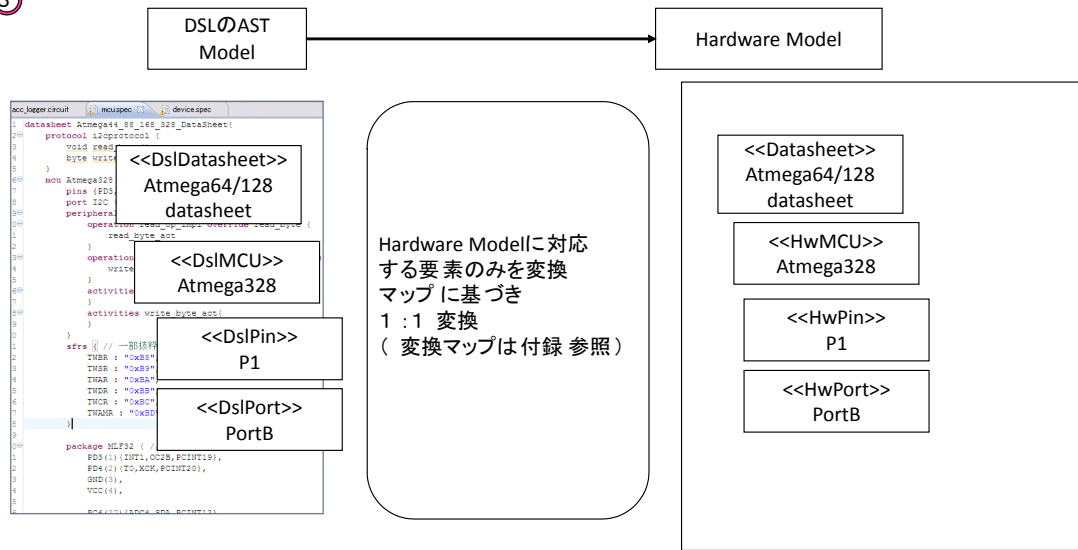


図 5.9: ハードウェア部品 DSL (MCU) の DSL からハードウェア部品モデルへの変換

④

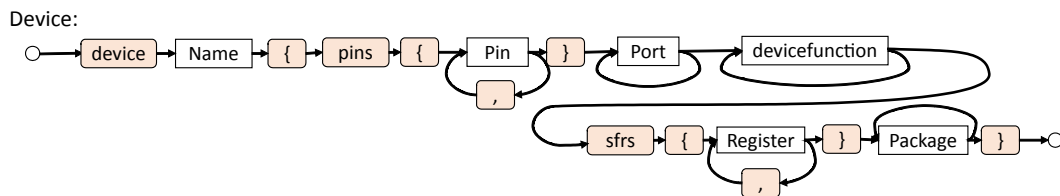


図 5.10: ハードウェア部品 DSL (デバイス) の定義 (一部抜粋)

④

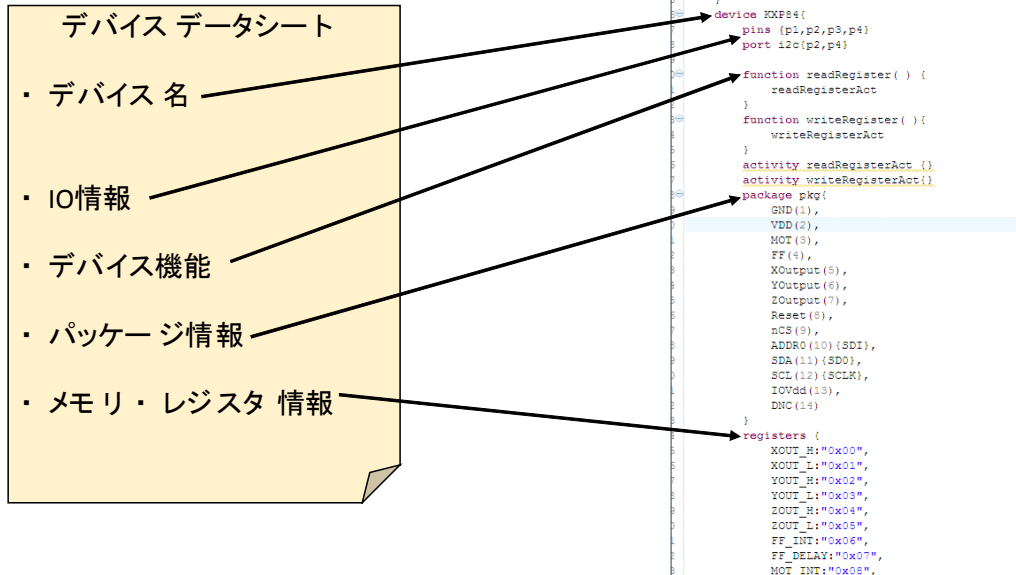


図 5.11: ハードウェア部品 DSL (デバイス) の入力例

ブロック図の入力はブロック図を図形式で入力できるようグラフィカル DSL で行う。ブロック図 DSL の定義を図 5.13 に示す。

ブロック図は、ブロック（四角）と関連線（線）からなる。ブロックとして、MCU またはデバイスを記述し、それらの間の接続を関連線で示す。また接続に用いられるプロトコル名を関連線に付与する。

ブロックはハードウェアメタモデルの HwComponent として、関連線は HwComponentConnection として定義する。

⑤

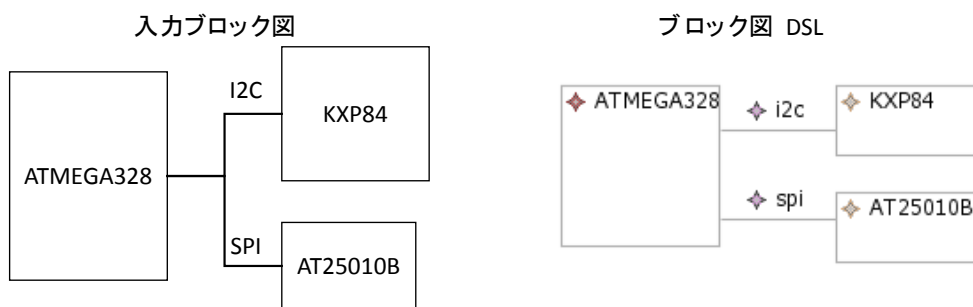


図 5.14: ブロック図 DSL の入力例

ブロック図 DSL の入力例を図 5.14 に示す。ブロック図の形式のまま記述するため、入力情報とブロック図 DSL との差異はない。

⑤

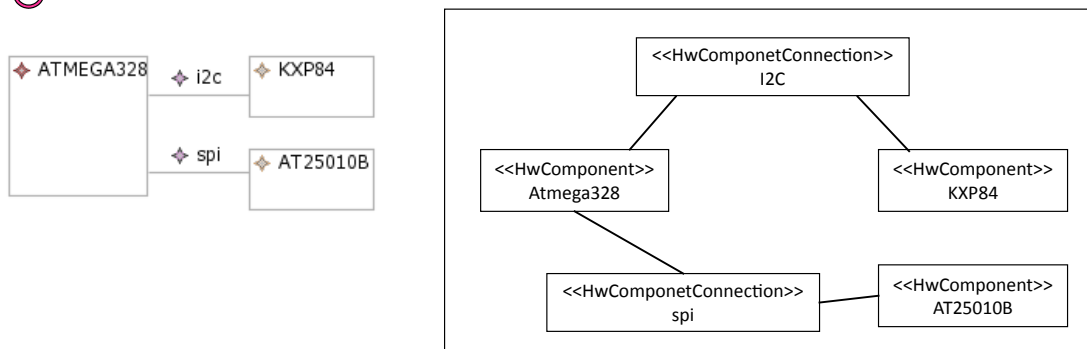


図 5.15: ブロック図 DSL からブロック図モデルへの変換

ブロック図 DSL からハードウェアモデルへの変換を図 5.15 に示す。ブロック図

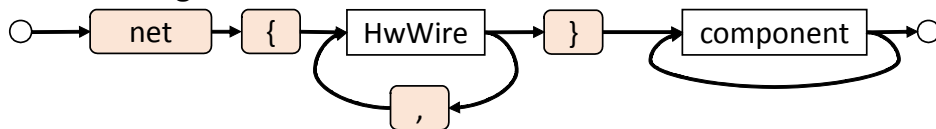
DSLのメタモデルはハードウェアメタモデルを用いているため、入力したDSLは直接ハードウェアモデルにマッピングされる。

5.2.4.3 回路図 DSL

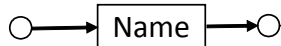
回路図DSLは、ハードウェア部品とその電氣的な結線情報を入力するものである。回路図は多くの回路図エディタでサポートされているネットリストエクスポート機能を利用し、ネットリストをスクリプトを使いDSLにテキスト変換し読み込む。

⑥

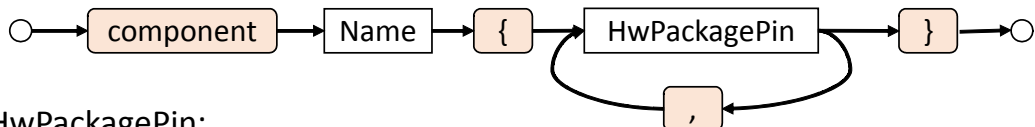
HwCircuitDiagram:



HwWire:



component:



HwPackagePin:

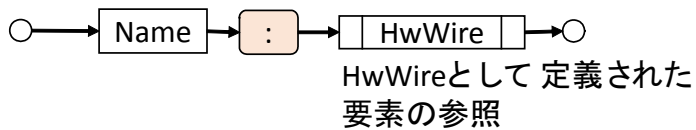


図 5.16: 回路図 DSL の定義

図5.16に回路図DSLの定義を示す。HwCircuitDiagramの定義では、前半のnetから}までが結線の定義、後半のComponentのがハードウェア部品の定義である。

結線の定義では指示詞netに続き大カッコで囲んだ中に、HwWireで結線名をカンマ区切りで列挙する。

一方ハードウェア部品である component では、指示詞 component に続きハードウェア部品名の記述し、大カッコ内にピン情報と結線情報を記述する。なお結線情報は、上記の結線で定義した要素の参照となっており、DSL 記述時には、予め結線の定義にて結線名を列挙しておくことで、参照することができる。

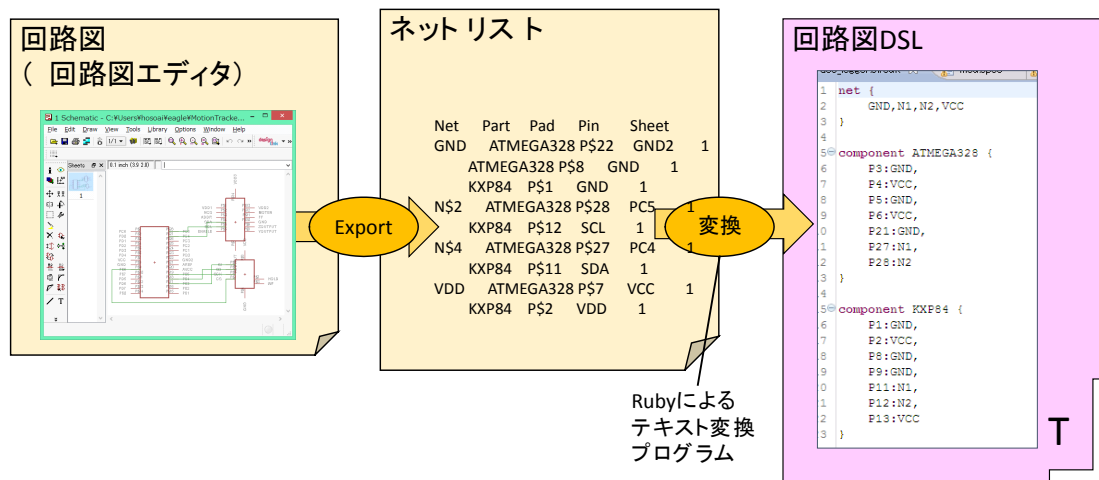


図 5.17: 回路図 DSL の入力例

図 5.17 に入力例を示す。回路図エディタから出力されたネットリストは、結線情報を表形式で表したもので、DSL で想定している表記法とは異なる。しかしながら、情報量としてはほぼ同等であるため、ネットリストから DSL へのテキスト変換を行うスクリプトを作成した。

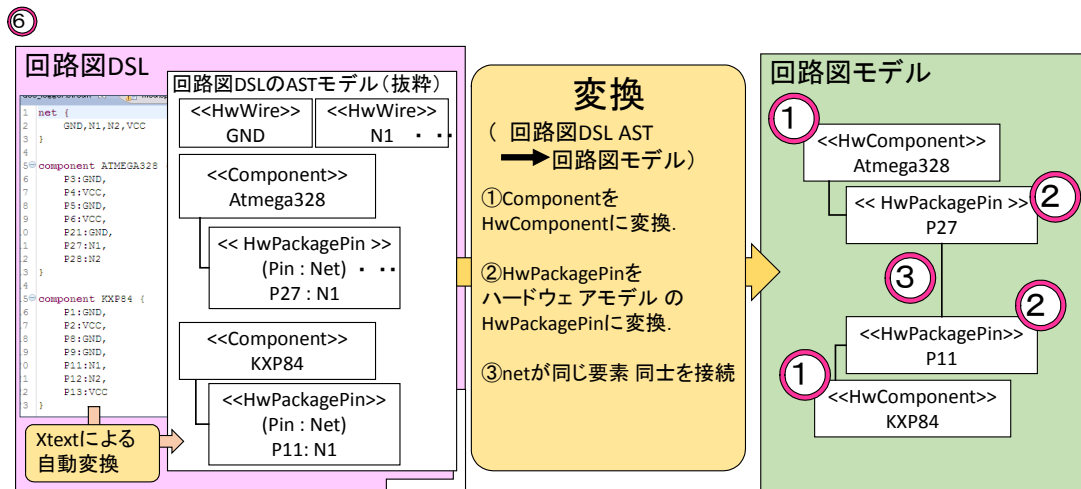


図 5.18: 回路図 DSL から回路図モデルへの変換

図 5.18 に回路図モデルへの変換を示す。

5.2.5 ハードウェアモデルの結合

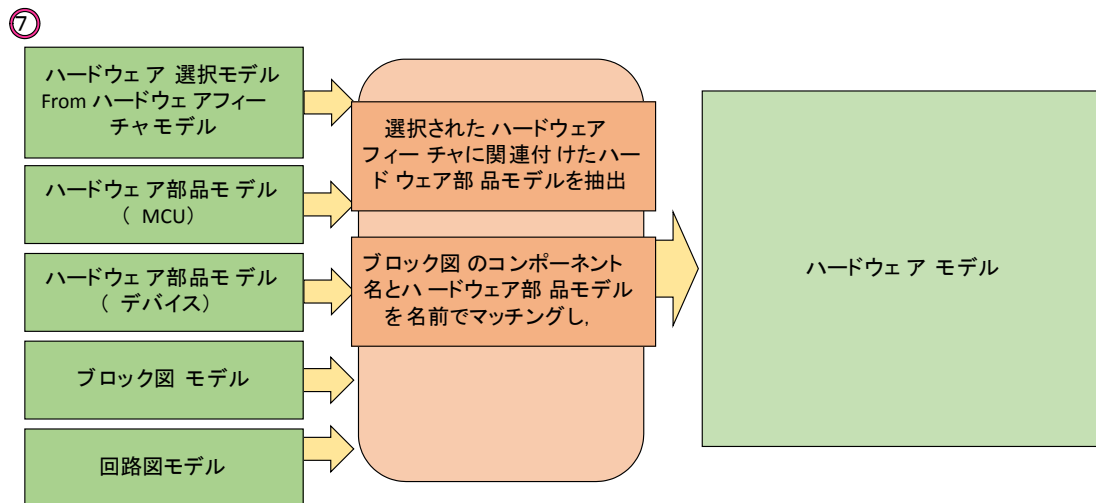


図 5.19: ハードウェアモデルの結合

5.2.6 ゲートウェイドライバ導出の変換定義

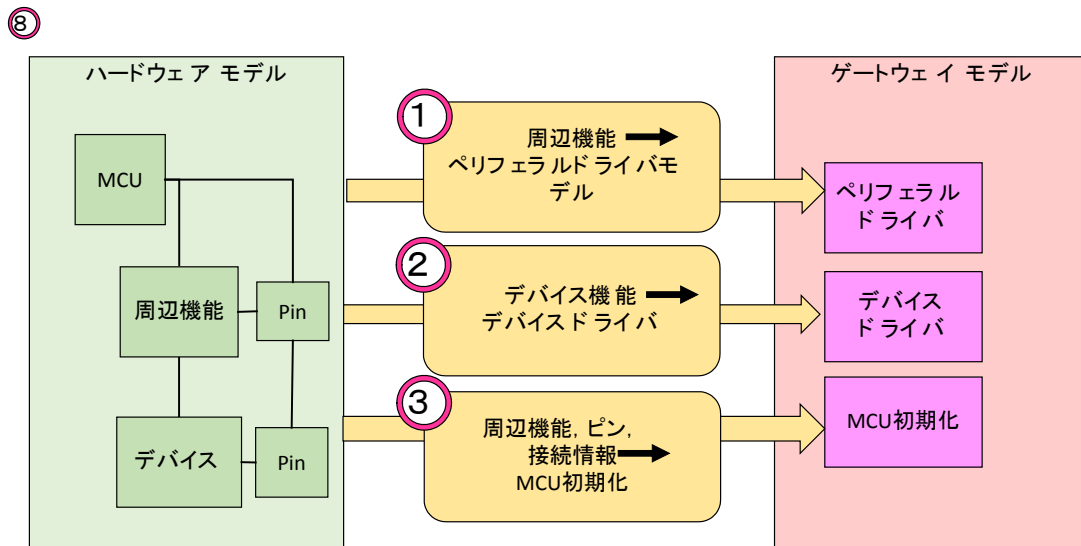


図 5.20: ハードウェアモデルからゲートウェイモデルへの変換の概要

ゲートウェイドライバの開発では、制御対象となるハードウェアに関する様々な情報を参照しながら開発を行う必要がある。主に用いるハードウェアに関する情報として以下のものが挙げられる。

- ハードウェア部品の情報（データシート）
- ハードウェア構成（ブロック図，回路図）

前節のハードウェアフィーチャモデルのハードウェアフィーチャはそのフィーチャを実現するハードウェア部品のハードウェアモデル要素に関連付けられており、ハードウェアプロダクトに必要なハードウェアフィーチャ群を選択することによって、そのハードウェアプロダクトに必要な部品の集合を得ることができる。これにブロック図や回路図といったハードウェア構成情報を付与し、部品単位に個別に定義されていたハードウェアモデル要素間に関連を結び、一つのハードウェアモデルを導出する。

ハードウェアモデルからは、そのハードウェアプロダクトを制御するゲートウェイモデルの導出を行う。ハードウェアモデルがハードウェアの視点で定義されてい

るのに対し、ゲートウェイモデルはソフトウェアの視点のモデルであるため、ハードウェアモデルからゲートウェイモデルへの変換には、ハードウェアからソフトウェアへの視点変換が必要となる。本章では、ハードウェアモデルの定義並びに、ハードウェアモデルからゲートウェイドライバへの変換について詳細に述べる。

5.2.6.1 モデル変換

モデル変換の手順をデバイスドライバの導出、ペリフェラルドライバの導出、初期化手順の導出と順に説明する。

デバイスドライバの導出 図5.21にハードウェアモデルからゲートウェイモデルのデバイスドライバの部分の導出を示す。入力としてハードウェアモデルで定義したブロック図、デバイスのデータシート、MCUのデータシートの各要素を用いて、ゲートウェイモデルのデバイスドライバ部分を導出する。

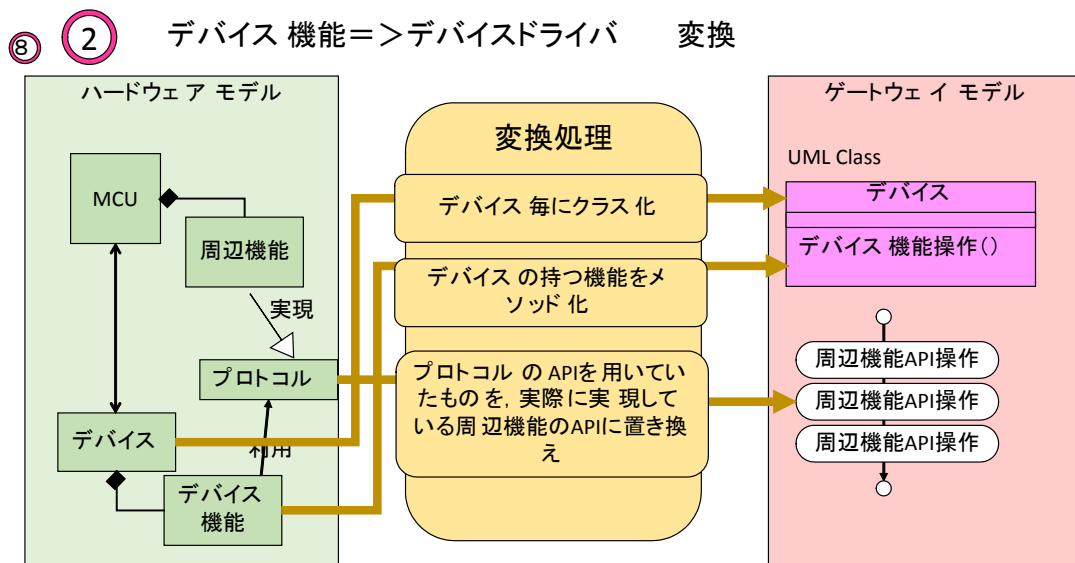


図 5.21: デバイスドライバの導出

まず、ブロック図 (HwBlockDiagram) に含まれる、HwDevice 要素を順に取り出し、デバイスドライバのクラスに相当する DeviceBroker に変換する。続いて、HwDevice の持つデバイス機能 (HwDeviceFunction) とその機能に対応する振舞

いの Activity を抽出し、それぞれ Operation と Activity に変換する。HwDeviceFunction と Operation はメソッドのシグネチャ、Activity は実装に相当する。

Activity には具体的な操作手順である Action の要素が含まれる（図では例としてメソッド呼び出しを表す CallOperationAction を用いている）。MCU とデバイス間の接続プロトコルの API は統一されたものはないため、仮想的な API を用いている。このため変換時に実際に MCU で定義されるペリフェラルドライバの API に読み替える必要がある。MCU のペリフェラルドライバの定義時には、この仮想 API インタフェースとして継承する形で実装を行うことで、この対応を関係付けている。以後同様にデバイスごとに変換処理を行う。

ペリフェラルドライバの導出 ペリフェラルドライバは、MCU のライブラリとして提供されている場合は、そのライブラリを DeviceBroker 要素として定義して用いる。ここでは提供されていない場合の導出手順を解説する。図 5.22 にペリフェラルドライバの導出手順を示す。

HwBlockDiagram で利用されている周辺機能（HwPeripheral）を抽出して、DeviceBroker に変換する。また、デバイスドライバの場合と同様に HwPeripheralFunction と Activity を Operation, Activity に変換する。HwPeripheral の Activity には周辺機能の操作に必要なレジスタの読み書きが必要となるが、通常の変数アクセスと多少扱いが異なるため、あらたに Action を追加した（ReadRegisterAction, WriteRegisterAction）。操作対象となるレジスタは MemoryBroker として定義する。

⑧ ① 周辺機能=>ペリフェラルドライバモデル 変換

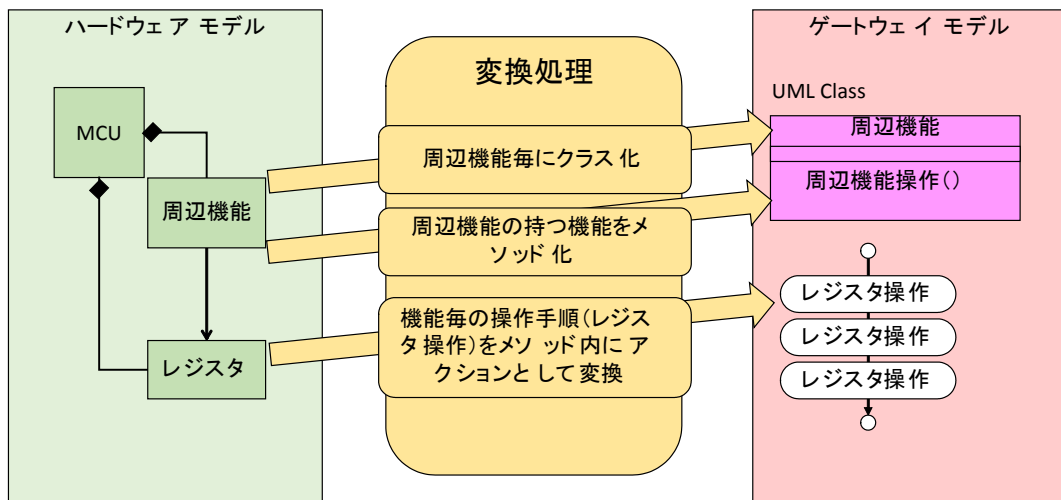


図 5.22: ペリフェラルドライバの導出

初期化処理の導出 初期化処理の導出を図 5.23 に示す。まず HwBlockDiagram にて利用されている周辺機能の抽出を行い、その周辺機能の初期化手順を MCU の初期化メソッドとして変換する。続いて HwCircuitDiagram よりその周辺機能が利用している IO の抽出を行い、その IO がどのように利用されているか確認する。IO の利用方法に合わせた設定処理を MCU の初期化メソッドに追加する形で変換する。

⑧ ③ 周辺機能, ピン , 接続情報 => MCU初期化 変換

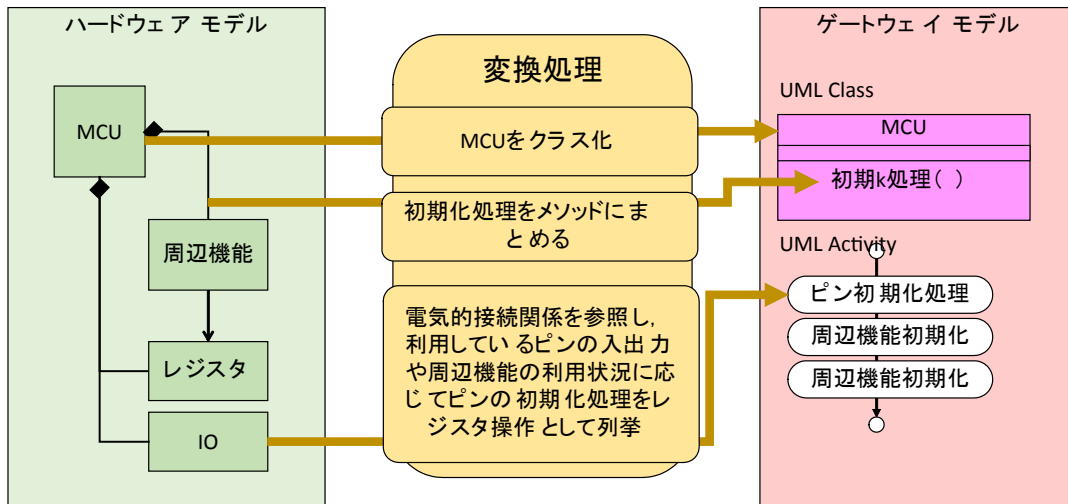


図 5.23: MCU 初期化処理の導出

5.2.6.2 コード生成

変換したソフトウェアモデルより、コード生成を行う。コード変換の定義は、コードテンプレートにより行う。ここではC言語への変換定義を例とする。C++やJava等を利用する場合には、そのコードテンプレートを用意する。

まず DeviceBroker ごとに一つのヘッダ・ソースファイルを生成する。外部から利用するメソッドは、ヘッダにエクスポート宣言する形で生成する。ソースファイルでは、Operation 要素を関数の定義として変換し、Activity に含まれる Action 要素を順次コードとして変換する。

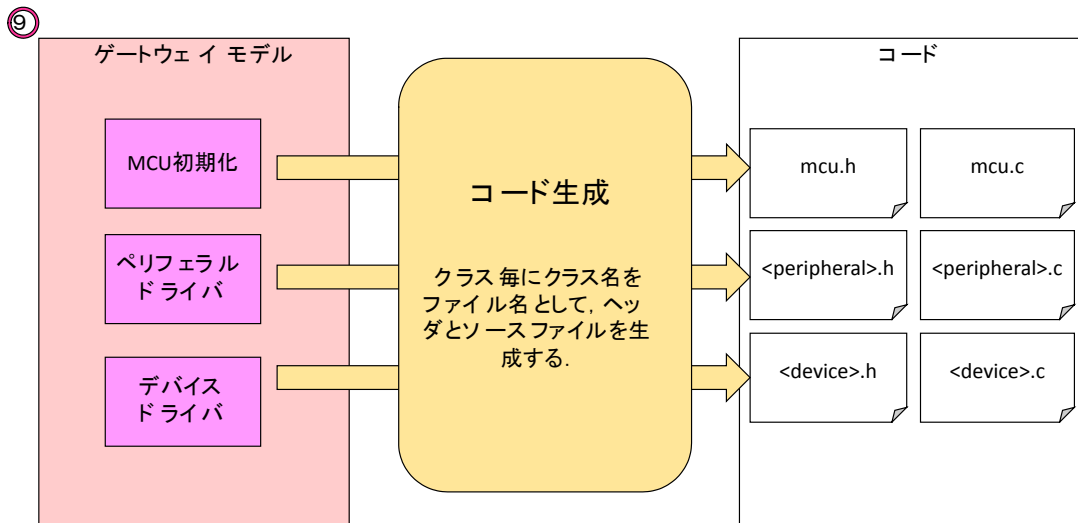


図 5.24: ゲートウェイモデルからゲートウェイドライバのコード生成

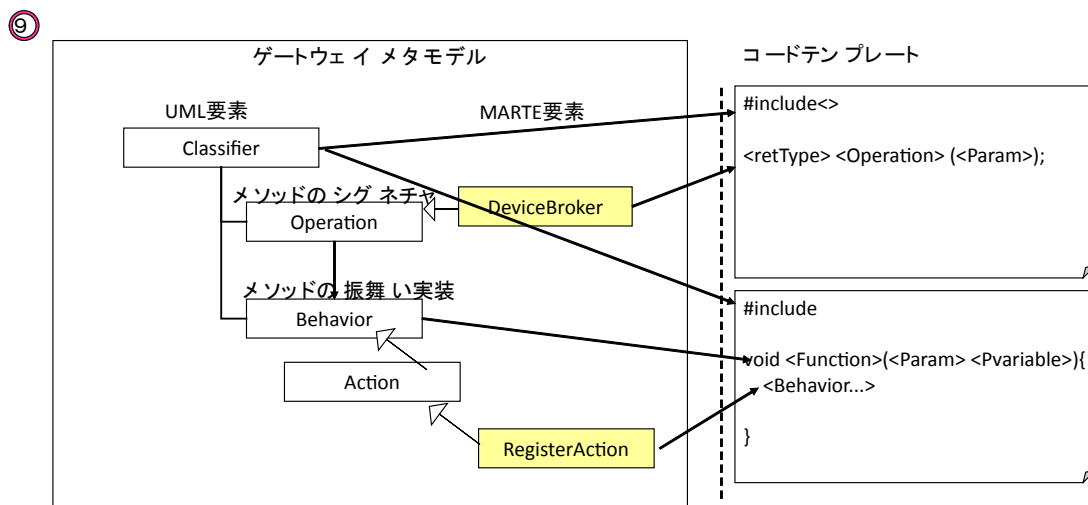


図 5.25: コードテンプレート例

第 6 章

評価

3章ではシステムプロダクトラインに関して問題を取り上げ、それぞれの分析を行った。本章では、提案手法の実装を行い、例題を通じて提案が問題を解決できているか評価を行う。

6.1 提案手法の実装

本研究の実証評価のために提案手法の実装を行う。

6.1.1 実装環境

提案システム全体のプラットフォームとして Eclipse[21] を用いる。Eclipse は主に Java の開発に用いられる統合開発環境でプラグインを追加することにより様々な言語への対応や機能拡張ができる。以下に述べる EMF[22], Xtext[24], GMF[25] もまた Eclipse のプラグインとして開発され提供されており、本提案も Eclipse のプラグインの形で実装を行う。

表 6.1 に今回の実装に用いた環境を示す。Papyrus UML[23] は EMF を基盤とした Eclipse 上の UML エディタで、特にプロファイルを扱う機能が充実している。MARTE や SysML など様々なプロファイルが含まれる。

Xtext は Domain Specific Language(DSL) を作成するためのフレームワークである。Xtext 言語で言語定義から、言語のメタモデルとエディタ、コード生成テン

プレートが Eclipse Plugin として生成される。

Xtend は Xtext で作成された Java の軽量言語である。Xtext のフレームワークの中でも多くの箇所で用いられており、モデル変換やコードテンプレートなどモデル駆動のために必要な機能が言語機能として実装されている。Xtend で記述された DSL は Java のソースコードに変換されるため、既存の Java ソフトウェアとの親和性も高い。

Graphical Modeling Framework (GMF) は、グラフィカル DSL を作成するためのフレームワークである。EMF のメタモデルを元にグラフィカルエディタを作成でき、入力されたモデルは EMF のインスタンスモデルとして利用することができる。

| Name | Version |
|----------------------------------|-------------------|
| Eclipse Modeling Tools[21] | Kepler(4.3.1) SR1 |
| Eclipse Modeling Framework[22] | 1.1.1 |
| Papyrus UML[23] | 0.10.1 |
| Xtext[24] | 2.4.3 |
| Xtend[24] | 2.4.3 |
| Graphical Modeling Framework[25] | 1.7.0 |

表 6.1: 使用環境

6.1.2 2層フィーチャメタモデルの実装

システムプロダクトラインの2層フィーチャモデルやハードウェアモデル、ゲートウェイモデルのモデル定義には、Eclipse Modeling Framework(EMF)を用いた。Eclipse Modeling Framework(EMF)はOMGのMDAのMeta Object Facility(MOF)を元に実装されたモデル駆動開発のためのフレームワークで、メタモデルの定義やモデル駆動技術のための基盤を提供する。モデル駆動のための多くのツールが本フレームワークを元に作られている。

フィーチャモデルのメタモデルもまたEMFのメタメタモデルであるEcoreを用いて実装されている(Ecore Feature Model)。フィーチャモデルのメタモデルは、Ecore Feature Modelのメタモデルを基盤として拡張を加えた。

6.1.3 ハードウェアメタモデル, ゲートウェイメタモデルの実装

ハードウェアモデルとゲートウェイモデルのメタモデル定義は Papyrus UML に含まれる MARTE プロファイルのメタモデルを元に, 本研究の拡張を加えた.

6.1.4 DSL の実装

ハードウェア部品 DSL と回路図 DSL の言語定義は, Xtext を用いて定義し, 言語メタモデルとエディタを生成した. また, 回路図エディタからエクスポートするネットリストから回路図 DSL への変換には, Ruby で作成した簡易なテキスト変換プログラムを用いた. ハードウェア部品 DSL からハードウェア部品モデルと回路図 DSL から回路図モデルへの変換は, Xtend にて DSL の言語メタモデルからハードウェアメタモデルへの変換ルールを記述することで行った.

ブロック図 DSL はグラフィカル DSL であるため, Graphical Modeling Framework を用いて定義を行った. ブロック図 DSL のメタモデルは, ハードウェアメタモデルを利用する. このため, 上記のような DSL からモデルへの変換は必要なく, DSL で入力した形のままモデルが作成した.

6.1.5 モデル変換の実装

ハードウェアフィーチャモデル, ハードウェア部品モデル, ブロック図モデル, 回路図モデルを結合しハードウェアモデルを得るプロセスは, Xtend にて定義した. ハードウェアモデルからゲートウェイモデルへの変換工程も Xtend を用いて定義した.

6.1.6 コード生成の実装

ゲートウェイモデルからゲートウェイドライバへのコード生成もまた, Xtend にて定義した. ゲートウェイドライバは C 言語を想定しており, C 言語のソースコードのテンプレートとヘッダファイルのためのテンプレートを作成した.

6.1.7 モデル駆動型システムプロダクトライン開発環境としての統合

これらのモデル操作のプロセスに加え、GUI 上からモデルファイルの選択や変換の指示などモデルフローの操作を指示できるよう、インタフェースを Eclipse のビュー Plugin として作成した。

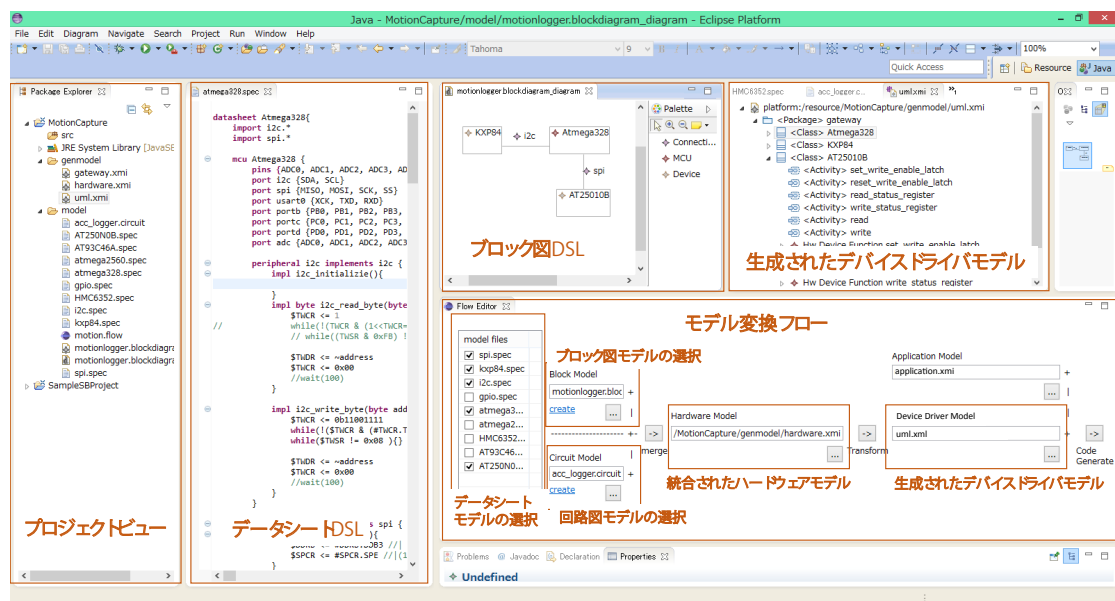


図 6.1: 統合環境の実行画面

実装した統合環境の実行画面を図 6.1 に示す。作成した環境は大きくプロジェクトビュー、モデルの入力部 (図中のデータシート DSL, ブロック図 DSL) と出力確認部 (図中の生成されたデバイスドライバモデル)、モデル変換フローに分けられる。プロジェクトビューにはプロジェクトに含まれるモデルやファイルを表示する。モデルの入力はデータシート DSL, ブロック図 DSL, 回路図 DSL に応じた三種類の DSL エディタを用いた。モデル変換フローでは入力したモデルを選択し、ハードウェアモデルの統合、ゲートウェイドライバモデルへの変換、コード生成といった各変換作業を行った。生成されたモデルは出力ビューで確認した。

6.2 例題システム

本提案の実現可能性の実証と評価のために簡易なモーショントラッキングシステムを例に本手法の適用を行う。モーショントラッキングシステムは、装着者の身体の動きを取得し、様々なサービスを提供するウェアラブルシステムである。主に装着部位の加速度を取得し、動作を推定する。加えて角速度や方位を取得することでより高精度な行動測定を行うものもある。

システムの製品系列として図 6.2 のようなものを考える。

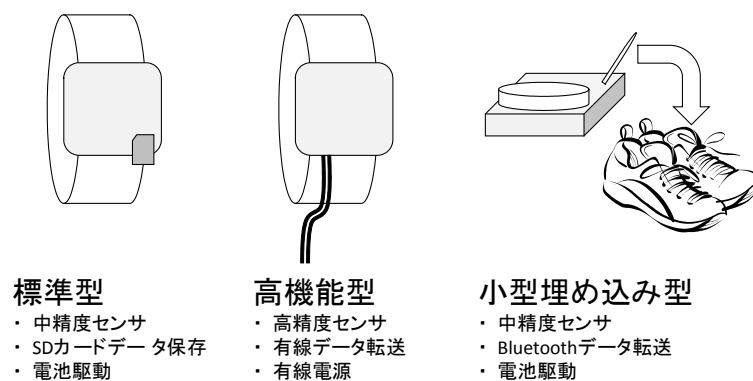


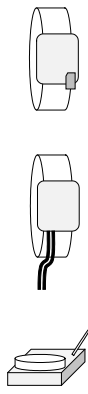
図 6.2: 例題システムの製品系列

標準型は腕時計型のデバイスで日常生活での行動トレースを主とする。取得したセンサデータはSDカードに保存し、PC上のアプリケーションにて取得したデータの分析を行う。

高機能型は、スポーツ計測など厳密なセンサデータを必要とするアプリケーションを対象とする。分析用のPCと有線接続し、逐次センサデータを転送する。

小型埋め込み型は、靴などに埋め込んだ形で利用する。取得したセンサデータはBluetoothでスマートフォンなどに転送し、スマートフォン上のアプリケーションで分析処理を行う。

図 6.3 に上記で示したハードウェアプロダクトの上で動作するアプリケーションの例を示す。



| ハードウェアプロダクト | ソフトウェアプロダクト |
|-------------|-------------|
| 標準型 | 日常生活計測 |
| | 睡眠計測 |
| 高機能型 | スポーツ計測 |
| | 研究計測 |
| 小型内蔵型 | ウォーキング計測 |
| | 介護計測 |

図 6.3: アプリケーション例

標準型のアプリケーション例として、日常生活での歩行やジョギングなどの行動をトレースや、睡眠時の身体の動きを計測することで睡眠状況を分析するものがあげられる。

高機能型のアプリケーション例として、スポーツの姿勢矯正のために詳細な動作計測を行う。また、別アプリケーション例として、映像作成に向けたモーションキャプチャーが挙げられる。

小型埋め込み型のアプリケーション例では、靴などに埋め込み腕時計型と同様に日常生活での行動トレースを行い、測定データはBluetoothによりスマートフォンに転送する。また別のアプリケーション例として、要介護者向けに転倒を検知し、介護者のスマートフォン等に通知する。

6.3 提案手法の適用

導出例は、ハードウェアプロダクトとして標準型と高機能型をを想定する。

6.3.1 システムプロダクトライン

例題システムの図 6.2 より想定するモーショントラッキングシステムのハードウェアの製品系列を考える。

今回対象とする三種類のハードウェアプロダクトを包括するハードウェアフィーチャモデルを図6.4に示す。

またソフトウェアプロダクトを包括するソフトウェアフィーチャを図6.5に示す。

ソフトウェアフィーチャモデルからハードウェアフィーチャモデルへの依存関係を図6.6に示す。

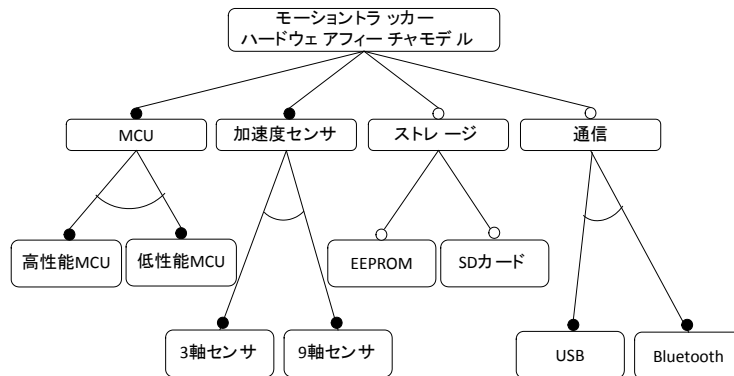


図 6.4: ハードウェアフィーチャモデル

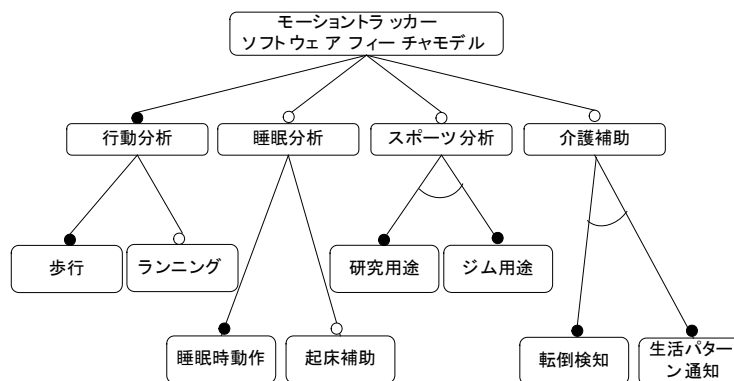


図 6.5: ソフトウェアフィーチャモデル

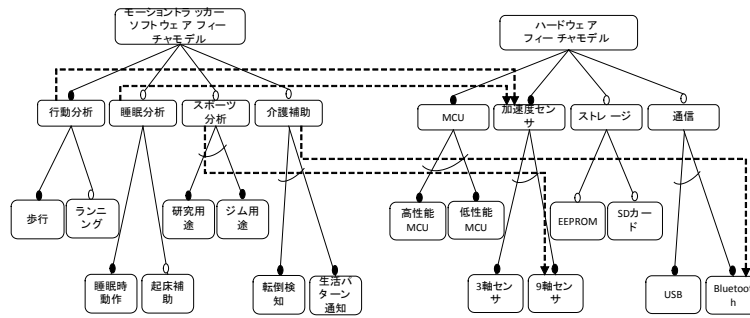


図 6.6: ソフトウェアフィーチャとハードウェアフィーチャの依存関係

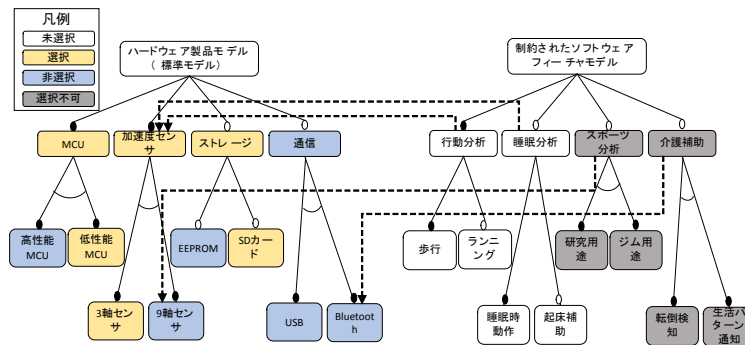


図 6.7: 導出されたハードウェアプロダクトと、制約されたソフトウェアフィーチャモデル

ここでは、5.1.2章の手順に従い基いてハードウェアフィーチャモデルから標準型のハードウェアプロダクトを導出し、その依存関係に基づきソフトウェアフィーチャモデルに制約加える。その結果を図 6.7 に示す。

6.3.2 ゲートウェイドライバの導出

ゲートウェイドライバの導出例は、ハードウェアプロダクトとして図 6.2 の標準型と高機能型を想定する。以後、標準型と高機能型それぞれにおけるハードウェア情報からゲートウェイドライバ導出までの流れを示す。

| モデル | 部品名 | 概要 |
|------|-----------------|----------------|
| 標準型 | ATmega328 [26] | Atmel 8bit MCU |
| | KXP84[27] | 3 軸加速度センサ |
| | AT25010B[28] | 1kByte EEPROM |
| 高機能型 | ATmega2560 [29] | Atmel 8bit MCU |
| | MPU9150 [30] | 9 軸加速度センサ |
| | FT232R [31] | USB-Serial 変換 |

表 6.2: ハードウェア部品一覧

ハードウェア部品情報の入力 表 6.2 に標準型と高機能型のハードウェアプロダクトで用いるハードウェア部品の一覧を示す.

MCU は標準型では Atmega328, 高機能型では Atmega2560 を用いる. 同じ Atmel 社の MCU であるが, 後者の方が IO 数やメモリ, 周辺機能が多く, 高クロックで動作する. 加速度センサは KXP84 と MPU9150 を用いる. KXP84 が 3 軸の加速度センサ (X,Y,Z 軸の加速度) であるのに対し, MPU9150 では, XYZ 軸に加えて, ジャイロによる角速度センサと地磁気センサを内蔵した 9 軸のセンサである. 標準型では測定データの一時保存用に EEPROM として AT25010B を用いる. 一方高機能型では FT232R でシリアルデータを USB 経由で PC に逐次転送する.

標準型

Atmega328

Atmel Atmel 8-bit Microcontroller with 4KB/16/32KBytes In-System Programmable Flash
 ATmega48A; ATmega48PA; ATmega88A; ATmega88PA; ATmega168A; ATmega168PA; ATmega328; ATmega328P

Features

- High Performance, Low Power Atmel AVR[®] 8-bit Microcontroller Family
- Advanced RISC Architecture
 - 131 Powerful Instructions - Most Single Clock Cycle Execution
 - 31 x 8 General Purpose Working Registers

Kionix ± 2g Tri-Axis Digital Accelerometer Specifications
 PART NUMBER: KXP84-2050 Rev. 2 Mar 07

Application Schematic

AT25010B

Atmel SPI Serial EEPROM
 1K (128x8)
 2K (256x8)
 4K (512x8)

Atmel AT25010B
 Atmel AT25020B
 Atmel AT25040B

Description
 The Atmel[®] AT25010B/25020B/25040B provides 1024/2048/4096 bits of serial electrically erasable programmable read-only memory (EEPROM) organized as 128/256/512 words of 8 bits each. The device is optimized for use in many industrial and consumer

高機能型

Atmega2560

Atmel Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V
 8-bit Atmel Microcontroller with 16384Kb In-System Programmable Flash

Features

- High Performance, Low Power Atmel AVR[®] 8-bit Microcontroller
- Advanced RISC Architecture
 - 131 Powerful Instructions - Most Single Clock Cycle Execution
 - 31 x 8 General Purpose Working Registers
 - 31 x 8-bit Registers
 - 32 16-bit Timers/counters w/ 16-bit Prescaler
 - On-Chip Analog-to-Digital Converter
 - High Endurance Non-Volatile Memory Elements

DATASHEET

InvenSense MPU-9150 Product Specification Document Number: PS-MPU9150A-1
 Revision: 4.3 Release Date: 9/18/2013

7.5 Block Diagram

MPU-9150

FTDI Chip FT232RL USB UART IC Datasheet Version 2.10 Clearance No.: FTDI# 2B
 Document No.: FT_000053

3 Device Pin Out and Signal Description

3.1 28-LD SSOP Package

図 6.8: データシート

図 6.8 にハードウェア部品のデータシートの一部抜粋を示す。データシートは、表 6.2 の参照に示した URL より、オンラインから入手する。

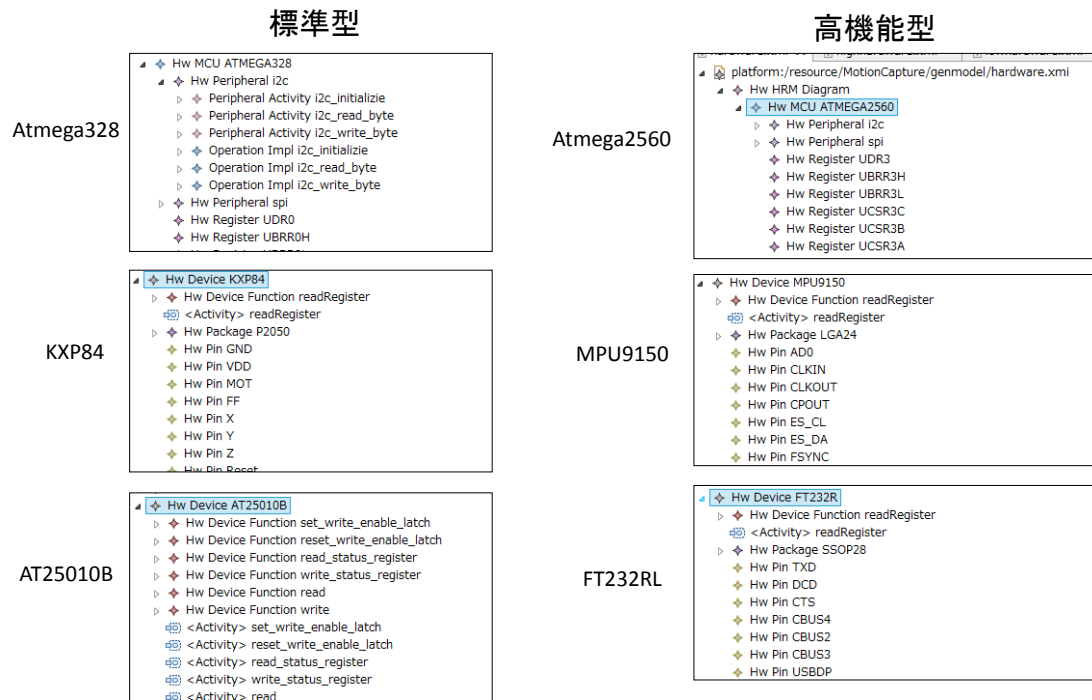


図 6.10: ハードウェア部品モデル

DSL で入力したハードウェア部品の情報は、Xtext により DSL の構文木モデルに自動変換される。DSL の構文木モデルは、5.2.4.1 に示した手順に従い、ハードウェアメタモデルに沿うハードウェア部品モデルに変換される。図 6.10 に変換されたハードウェア部品モデルの一部抜粋を示す。

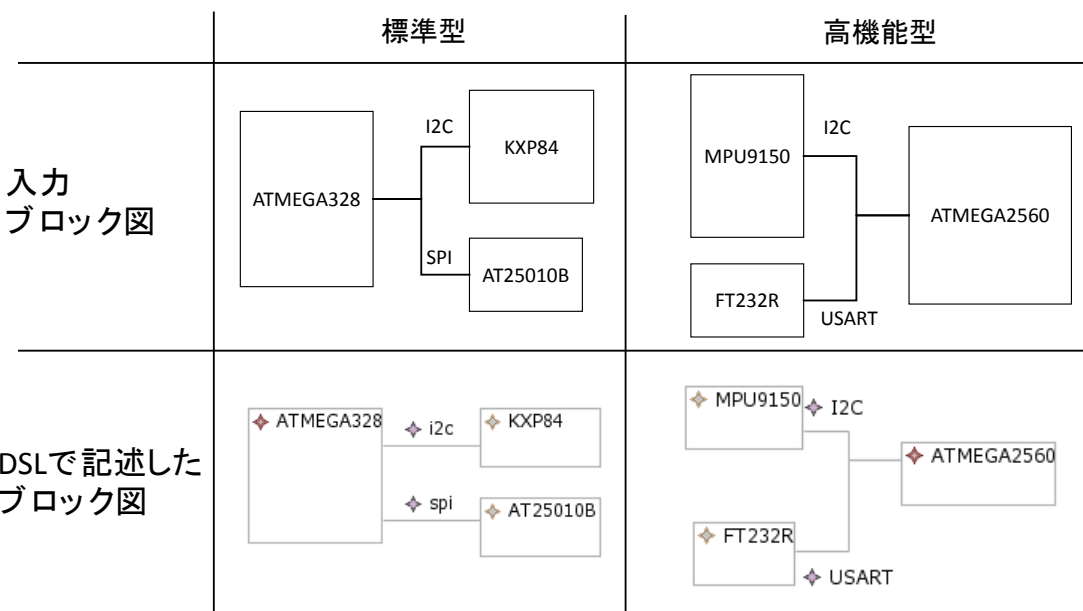


図 6.11: ブロック図と入力したブロック図 DSL

ブロック図の入力 図 6.11 に入力に用いたブロック図と入力したブロック図 DSL を示す。

ブロック図 DSL は、ブロック図を図形式で入力するため入力情報と DSL との乖離は少ない。ただし、入力するブロック図によっては、MCU とデバイス間の接続プロトコルについて言及していないものもあるため、そのような場合は、入力者が付与する必要がある。今回は接続プロトコルは予め提示されているものとする。提示されていない場合も、ハードウェア設計段階では接続プロトコルは決定していることが殆どである。

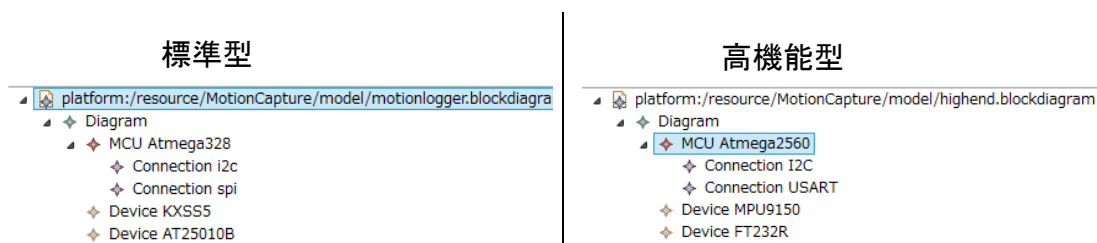


図 6.12: ブロック図モデル

ブロック図 DSL からブロック図モデルへの変換について図 6.12 に示す。ブロッ

ク図 DSL はハードウェアメタモデルに基いて定義されているため、5.2.4.3 で示した手順に従い入力された DSL はそのままブロック図モデルとして変換される。

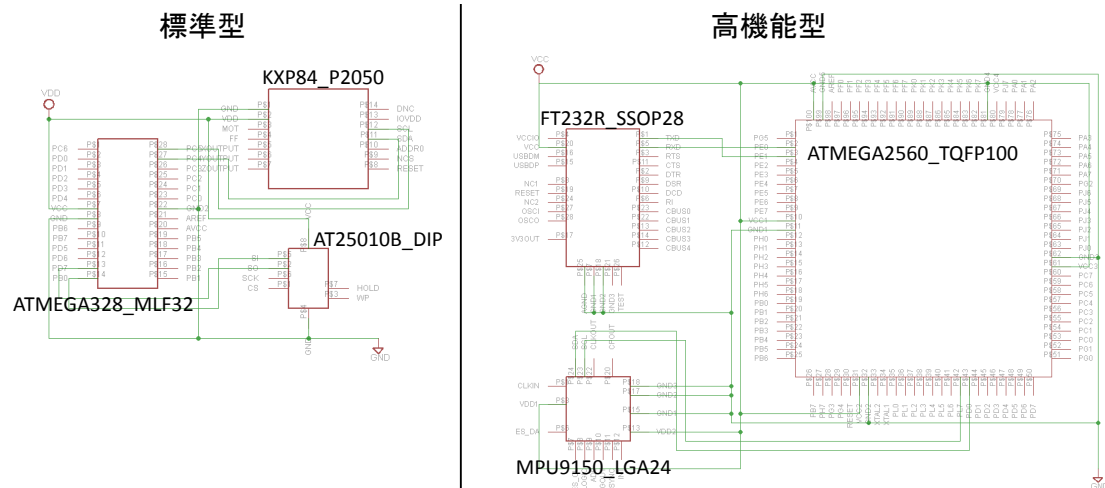


図 6.13: 回路図

回路図の入力 回路図の入力について示す。図 6.13 に入力に用いた回路図を示す。

回路図の入力には Eagle[45] という回路図エディタを用いた。Eagle で記述した回路図は、エディタのエクスポート機能を用いてネットリストを出力した。出力されたネットリストを図 6.14 に示す。図では不要なヘッダ部などは除去している。

標準型

| Net | Part | Pad | Pin | Sheet |
|------|-----------------|-------|------|-------|
| GND | ATMEGA328_MLF32 | P\$22 | GND2 | 1 |
| | ATMEGA328_MLF32 | P\$8 | GND1 | 1 |
| | KXP84_P2050 | P\$1 | GND | 1 |
| N\$1 | AT25010B_DIP | P\$2 | SO | 1 |
| | ATMEGA328_MLF32 | P\$13 | PD7 | 1 |
| N\$2 | ATMEGA328_MLF32 | P\$28 | PC5 | 1 |
| | KXP84_P2050 | P\$12 | SCL | 1 |
| N\$3 | AT25010B_DIP | P\$5 | SI | 1 |
| | ATMEGA328_MLF32 | P\$14 | PB0 | 1 |
| N\$4 | ATMEGA328_MLF32 | P\$27 | PC4 | 1 |
| | KXP84_P2050 | P\$11 | SDA | 1 |
| VDD | AT25010B_DIP | P\$8 | VCC | 1 |
| | ATMEGA328_MLF32 | P\$7 | VCC | 1 |
| | KXP84_P2050 | P\$2 | VDD | 1 |

高機能型

| Net | Part | Pad | Pin | Sheet |
|------|--------------------|--------|------|-------|
| GND | ATMEGA2560_TQFP100 | P\$11 | GND1 | 1 |
| | ATMEGA2560_TQFP100 | P\$32 | GND2 | 1 |
| | ATMEGA2560_TQFP100 | P\$62 | GND3 | 1 |
| | ATMEGA2560_TQFP100 | P\$99 | GND5 | 1 |
| | FT232R_SSOP28 | P\$18 | GND2 | 1 |
| | FT232R_SSOP28 | P\$25 | AGND | 1 |
| | MPU9150_LGA24 | P\$15 | GND1 | 1 |
| | MPU9150_LGA24 | P\$17 | GND2 | 1 |
| | MPU9150_LGA24 | P\$18 | GND3 | 1 |
| N\$1 | ATMEGA2560_TQFP100 | P\$3 | PE1 | 1 |
| | FT232R_SSOP28 | P\$1 | TXD | 1 |
| N\$2 | ATMEGA2560_TQFP100 | P\$2 | PE0 | 1 |
| | FT232R_SSOP28 | P\$5 | RXD | 1 |
| N\$4 | ATMEGA2560_TQFP100 | P\$42 | PL7 | 1 |
| | MPU9150_LGA24 | P\$23 | SCL | 1 |
| N\$7 | ATMEGA2560_TQFP100 | P\$43 | PDO | 1 |
| | MPU9150_LGA24 | P\$24 | SDA | 1 |
| VCC | ATMEGA2560_TQFP100 | P\$10 | VCC1 | 1 |
| | ATMEGA2560_TQFP100 | P\$100 | AVCC | 1 |
| | ATMEGA2560_TQFP100 | P\$31 | VCC2 | 1 |
| | ATMEGA2560_TQFP100 | P\$61 | VCC3 | 1 |
| | ATMEGA2560_TQFP100 | P\$81 | GND4 | 1 |
| | FT232R_SSOP28 | P\$20 | VCC | 1 |
| | MPU9150_LGA24 | P\$13 | VDD2 | 1 |
| | MPU9150_LGA24 | P\$3 | VDD1 | 1 |

図 6.14: ネットリスト

ネットリストは、配線名 (Net)、部品名 (Part)、端子名 (Pin)、シート (Sheet) といった表形式で記述されている。一方 DSL では、配線名の定義と部品の定義といった形式となっている。このため、表形式のネットリストをスクリプトプログラムにより DSL へ変換して入力した。

図 6.15 にスクリプトプログラムでネットリストから DSL に変換したものを示す。

標準型

```
net {
  GND, N1, N2, N3, N4, VDD
}
component ATMEGA328_MLF32 {
  GND2:GND,
  GND1:GND,
  PD7:N1,
  PC5:N2,
  PB0:N3,
  PC4:N4,
  VCC:VDD
}
component KXP84_P2050 {
  GND:GND,
  SCL:N2,
  SDA:N4,
  VDD:VDD
}
component AT25010B_DIP {
  SO:N1,
  SI:N3,
  VCC:VDD
}
```

高機能型

```
net {
  GND, N1, N2, N4, N7, VCC
}
component ATMEGA2560_TQFP100 {
  GND1:GND,
  GND2:GND,
  GND3:GND,
  GND5:GND,
  PE1:N1,
  PE0:N2,
  PL7:N4,
  PD0:N7,
  VCC1:VCC,
  AVCC:VCC,
  VCC2:VCC,
  VCC3:VCC,
  GND4:VCC
}
component FT232R_SSOP28 {
  GND2:GND,
  AGND:GND,
  TXD:N1,
  RXD:N2,
  VCC:VCC
}
component MPU9150_LGA24 {
  GND1:GND,
  GND2:GND,
  GND3:GND,
  SCL:N4,
  SDA:N7,
  VDD2:VCC,
  VDD1:VCC
}
```

図 6.15: 回路図 DSL

回路図 DSL では、定義されている配線の一覧と、部品の定義、接続されている部品の端子名と配線名で構成される。回路図 DSL はテキスト DSL であるため、Xtext により構文木モデルに自動変換されたのち、回路図モデルに変換した。変換した回路図モデルを図 6.16 に示す。

標準型

- platform:/resource/MotionCapture/genmodel/circuit.xmi
 - Hw Circuit Diagram
 - Hw Package ATMEGA328_MLF32
 - Pin GND2
 - Pin GND1
 - Pin PD7
 - Pin PC5
 - Pin PB0
 - Pin PC4
 - Pin VCC
 - Hw Package KXP84_P2050
 - Pin GND
 - Pin SCL
 - Pin SDA
 - Pin VDD
 - Hw Package AT25010B_DIP
 - Pin SO
 - Pin SI
 - Pin VCC
 - Hw Wire GND
 - Hw Wire N1
 - Hw Wire N2
 - Hw Wire N3
 - Hw Wire N4
 - Hw Wire VDD

高機能型

- platform:/resource/MotionCapture/genmodel/circuit.xmi
 - Hw Circuit Diagram
 - Hw Package ATMEGA2560_TQFP100
 - Pin GND1
 - Pin GND2
 - Pin GND3
 - Pin GND5
 - Pin PE1
 - Pin PE0
 - Pin PL7
 - Pin PD0
 - Pin VCC1
 - Pin AVCC
 - Pin VCC2
 - Pin VCC3
 - Pin GND4
 - Hw Package FT232R_SSOP28
 - Pin GND2
 - Pin AGND
 - Pin TXD
 - Pin RXD
 - Pin VCC
 - Hw Package MPU9150_LGA24
 - Pin GND1
 - Pin GND2
 - Pin GND3
 - Pin SCL
 - Pin SDA
 - Pin VDD2
 - Pin VDD1
 - Hw Wire GND
 - Hw Wire N1
 - Hw Wire N2
 - Hw Wire N4
 - Hw Wire N7
 - Hw Wire VCC

図 6.16: 回路図モデル

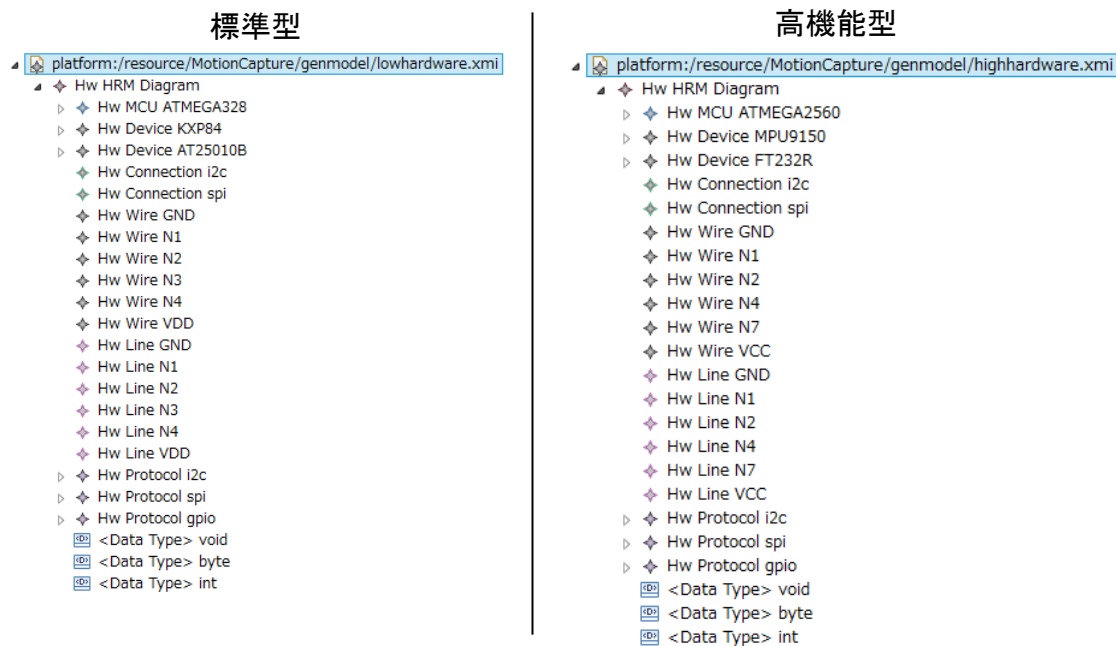
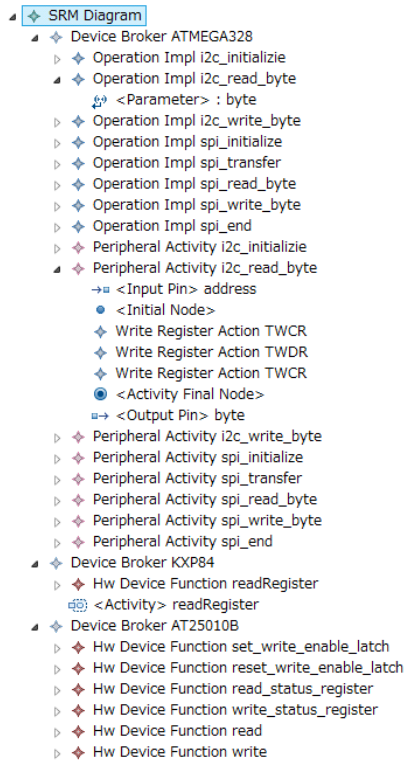


図 6.17: ハードウェアモデル

ハードウェアモデルの合成 5.2.5 に示した手順に従い、上記のハードウェア選択モデル、MCU とデバイスのハードウェア部品モデル、ブロック図モデル、回路図モデルを合成し、導出されたハードウェアプロダクトのハードウェア情報を全て含むハードウェアモデルを図 6.17 に示す。

標準型



高機能型

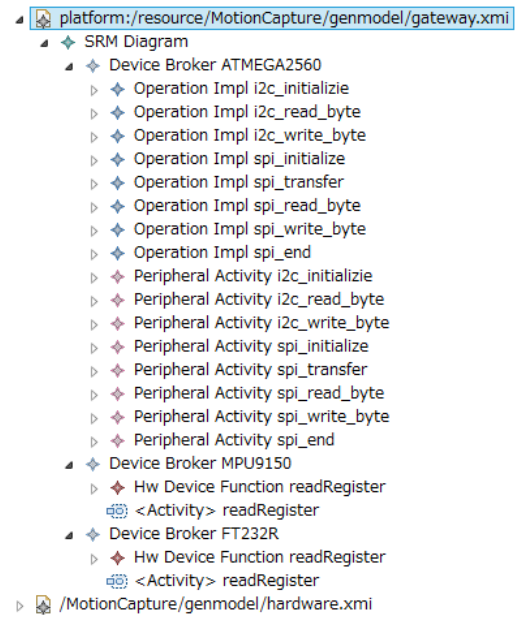


図 6.18: ゲートウェイモデル

ハードウェアモデルからゲートウェイモデルへの変換 5.2.6 で示した手順に従いハードウェアモデルから変換したゲートウェイモデルを図 6.18 に示す。

| 標準型 | | 高機能型 | |
|--|--|---|---|
| atmega328.h <pre>#ifndef __ATMEGA328_H__ #define __ATMEGA328_H__ void i2c_initialize(); void i2c_read_byte(); void i2c_write_byte(); void spi_initialize(); void spi_transfer(); void spi_read_byte(); void spi_write_byte(); void spi_end(); #endif</pre> | atmega328.c <pre>#include "atmega328.h" void i2c_initialize(){ } void i2c_read_byte(){ } void i2c_write_byte(){ } void spi_initialize(){ } void spi_transfer(){ } void spi_read_byte(){ } void spi_write_byte(){ } #endif</pre> | atmega2560.h <pre>#ifndef __ATMEGA2560_H__ #define __ATMEGA2560_H__ void i2c_initialize(); void i2c_read_byte(); void i2c_write_byte(); void spi_initialize(); void spi_transfer(); void spi_read_byte(); void spi_write_byte(); void spi_end(); #endif</pre> | atmega2560.c <pre>#include "atmega2560.h" void i2c_initialize(){ } void i2c_read_byte(){ } void i2c_write_byte(){ } void spi_initialize(){ } void spi_transfer(){ } void spi_read_byte(){ } void spi_write_byte(){ } void spi_end(){ } #endif</pre> |
| kxp84.h <pre>#ifndef __KXP84_H__ #define __KXP84_H__ void readRegister(); #endif EOF</pre> | kxp84.c <pre>#include "kxp84.h" void readRegister(){ } #endif EOF</pre> | mpu9150.h <pre>#ifndef __MPU9150_H__ #define __MPU9150_H__ void readRegister(); #endif EOF</pre> | mpu9150.c <pre>#include "mpu9150.h" void readRegister(){ } #endif EOF</pre> |
| at25010b.h <pre>#ifndef __AT25010B_H__ #define __AT25010B_H__ void set_write_enable_latch(); void reset_write_enable_latch(); void read_status_register(); void write_status_register(); void read(); void write(); #endif</pre> | at25010b.c <pre>#ifndef __AT25010B_H__ #define __AT25010B_H__ void set_write_enable_latch(); void reset_write_enable_latch(); void read_status_register(); void write_status_register(); void read(); void write(); #endif</pre> | ft232r.h <pre>#ifndef __FT232R_H__ #define __FT232R_H__ void readRegister(); #endif</pre> | ft232r.c <pre>#include "ft232r.h" void readRegister(){ } #endif EOF</pre> |

図 6.19: 生成されたゲートウェイドライバのコード

ゲートウェイモデルからゲートウェイドライバソースコードの生成 5.2.6.2 に示した手順に従い、ゲートウェイモデルから C 言語をソースコード生成したものを図 6.19 に示す。出力したファイルでは、振舞いモデルからのコード変換定義が足りていないため、関数のシグネチャ部分のコード生成のみを扱っている。

6.4 評価

3章では、以下のような問題を取り上げた。それぞれの項目に関して問題を解決できたか評価を行った。

- 組込みシステムプロダクトライン開発における問題
 - － ハードウェアとソフトウェアそれぞれのプロダクトライン
 - － ハードウェアフィーチャモデルとソフトウェアフィーチャモデルの依存関係
- ゲートウェイドライバ導出における問題点

- ハードウェアの多様性
- ハードウェア情報の多様性
- 開発手順の暗黙性

6.4.1 組み込みシステムプロダクトライン開発における問題

ハードウェアとソフトウェアの多様性の管理の問題について、本研究では2層フィーチャモデルを用いたシステムプロダクトラインを提案した。

6.4.1.1 ハードウェアとソフトウェアそれぞれのプロダクトライン

2層フィーチャモデルにより、ハードウェアの製品系列とソフトウェアの製品系列を独立して定義できるようになり、ハードウェアとソフトウェアそれぞれの構成管理を提案した。2層フィーチャモデルにより、ハードウェア、ソフトウェアそれぞれの関心事にもとづいてハードウェアフィーチャモデルとソフトウェアフィーチャモデルを構成することができた。

また、ハードウェアフィーチャとハードウェア部品モデルとの関連は、ハードウェアフィーチャモデルでのみ定義し、ソフトウェアフィーチャモデルとは関連を持たせない、これによりハードウェアとソフトウェアを独立に開発することができる。

6.4.1.2 ハードウェアフィーチャモデルとソフトウェアフィーチャモデルの依存関係

2層のフィーチャモデルに分けたが、一部のソフトウェアはハードウェアに依存するため、実現不可能なソフトウェアプロダクトが導出されてしまう恐れがある。またハードウェアフィーチャモデルとソフトウェアフィーチャモデルから導出される製品は、両モデルの組み合わせとなるため、膨大な製品が導出できてしまう。

そこで本研究では、2層フィーチャモデルのハードウェア・ソフトウェア間に依存関係を定義できるようにし、ハードウェア製品を導出後、ソフトウェアフィーチャモデルに対して制約を加えることで段階的に製品導出をする手法の提案を行っ

た。段階的に製品導出を行うことで、無効なソフトウェア製品の導出を排除するとともに、膨大となる製品系列の組み合わせを削減することができた。

フィーチャモデルを段階的に導出しない場合と、依存関係を考慮して段階的に導出した場合のフィーチャの組み合わせ数について評価を行った。今回用いたハードウェアフィーチャモデルとソフトウェアフィーチャモデルにおいて、両モデルを分離せず、全組み合わせで製品導出を行った場合、2592個の組み合わせが導出される。一方、本手法を適用し、段階的な導出を行った場合、組み合わせの数は648となる。

また今回のようにハードウェアプロダクトが3つに限定される場合であれば、標準型のハードウェアプロダクトでは6、高機能型では18、小型埋め込み型では18の計42の組み合わせに削減することが出来た。

6.4.2 ゲートウェイドライバ導出における問題点

ハードウェア情報からゲートウェイドライバ作成の問題について、本研究ではモデル駆動技術によりハードウェア情報からゲートウェイドライバまでの変換プロセスの定義を行うことで解決を行った。

6.4.2.1 ハードウェアの多様性

組込みソフトウェア開発では、多種多様なハードウェア製品を扱う必要がある。本研究では、5.2.3に示したように、組込みソフトウェア開発に必要なハードウェア情報を限定し、形式化することで、開発手順を明示化し、導出するプロセスの提案を行った。

6.4.2.2 ハードウェア情報の多様性

ハードウェア情報はメーカーの提供するデータシートやハードウェア設計から提供されるブロック図、回路図であるがこれらの情報のフォーマットは統一されていない。本研究では、ハードウェア情報の種類に応じたDSLを定義することにより、ハードウェア情報の入力を容易とするとともに、統一的な情報の入力が行えるようになった。

6.4.2.3 開発手順の暗黙性

ハードウェア情報を形式化，モデル化することにより，ハードウェア情報からゲートウェイドライバ導出までの手順をモデル変換として明示的に示すことができた。

第 7 章

関連研究

7.1 フィーチャモデルと製品導出

本提案では 2 層のフィーチャモデルを提案し，フィーチャモデルを分割するとともに，段階的に導出する手法を提案した．フィーチャモデルを分割する手法については，いくつかの研究がある．

7.1.1 フィーチャモデルの分割

当初フィーチャモデル [19] は，ドメイン分析のためのモデリング手法として提案されたが，徐々に再利用設計のためのモデリング手法としてもつかわれるようになり，設計や実装に関するフィーチャも記述されるようになってきた．Kang らは，再利用設計手法 FORM [20] の中で，フィーチャを capability, operating environment, domain technology, implementation technique の 4 つのレイヤにカテゴリわけし，フィーチャモデルを 4 つのレイヤに分けて記述する方法を提案している．このレイヤ分けは，あくまで思考のリファレンスとしてのカテゴリわけである．

ひとつのシステムが複数のプロダクトラインによって構成されているという考え方はマルチプロダクトラインと呼ばれる．一般にマルチプロダクトラインではプロダクトライン毎にフィーチャモデルを構築し，それらの間を関連付ける手法をとることが多いが，あるソフトウェアプロダクトラインの構成が変わると，他のソフトウェアプロダクトラインの異なる構成が必要となるため，依存関係管理

が複雑となる。

Rousenmuller ら [33] は、こうしたマルチプロダクトラインでの構成導出について提案している。また Kruegar ら [32] は、プロダクトラインがネスト構造を持つことがソフトウェアプロダクトラインを複雑化する問題を指摘している。本研究での2層フィーチャモデルは、こうした考え方をシステムプロダクトライン開発に拡張したものである。

マルチプロダクトラインはひとつのソフトウェアプロダクトライン中に内包されるサブシステムをプロダクトラインとしてとらえるものであるが、2層フィーチャモデルはシステムプロダクトラインにおけるハードウェアのプロダクトラインと、ソフトウェアのプロダクトラインに注目し、それをソフトウェア開発の視点から二つのフィーチャモデルとして整理した点に新規性がある。

7.1.2 段階的な製品導出

フィーチャモデル中の可変フィーチャの必要性を判断して特定の製品に必要となるフィーチャ群を決定し、それに基づき製品を導出する方法は広く使われているが、そうした製品導出のための決定作業を多段階に分けて行うアイデアも提案されている。

Czarnecki ら [33] は、フィーチャモデルに対する意思決定を、係るステークホルダーが時系列にそって順次行っていき、最終的なシステムの構成を決定する段階的構成の手法を提案している。White ら [34] は、フィーチャモデルの決定を多段にすると導出のための計算量が大きくなる等の問題を指摘している。この問題に対し、製品導出の多段化の形式モデルを作成し、ソルバを用いて製品導出の自動化を行っている。

2層フィーチャモデルの製品導出は、システムプロダクトライン開発におけるフィーチャモデルからの製品導出手順を提案した点、さらにハードウェア製品の導出を行うことで、その上に開発されるソフトウェア製品群に対する制約を明確化する手順を示した点などに特徴がある。

7.2 デバイスドライバの生成

本研究ではゲートウェイドライバの生成を行っているが、こうしたハードウェアとのインタフェースとなるソフトウェアを自動生成する研究も多い。なおこの節ではそうした研究の記述にあわせてデバイスドライバの生成として説明する。

例えば Marlet ら [38] は、Linux のビデオカードのデバイスドライバを生成するための DSL を開発している。Linux プラットフォームでは、ある程度デバイスドライバの形態が固まっており、デバイスもすべてブロックデバイスとして認識される。レジスタの記述等は本手法と共通する要素であり、彼らの手法では DSL から直接コードへ変換を行っている。我々の手法では、ハードウェアモデルとして扱い、ソフトウェアモデルと一緒に設計に用いる。Linux のデバイスドライバの扱いと組み込みのデバイスドライバ開発との違いによるものであると思われる。

こうしたデバイスドライバ生成をモデル駆動開発によって行う研究もある。

Hui ら [36] は、MPSoC (Multi Processor System on Chip) 用デバイスドライバを生成するモデル駆動開発環境 Me3D の研究を行っている。Me3D では、制御対象となるデバイスのハードウェア情報と機能定義、カーネルインタフェースの情報を用いてデバイスドライバの生成を行う。論文では、マルチメディアカードのデバイスドライバ生成を例に取り上げている。

デバイスの機能定義には C 言語かデバイス機能定義用の小規模な DSL を提供しており、必要となるデバイス機能の振舞い定義を行っている。

本研究と比較した際、カーネルインタフェースも用いており、OS との親和性が高いことが挙げられる。デバイスドライバの振舞い定義については、本研究と同様に DSL での入力を行っている。ハードウェアプラットフォームに関する情報は IP-XACT にて定義して用いている。しかしながら、DMA や割り込みといった特定のハードウェアのみを扱っている。これは、Linux で提供されているハードウェア抽象化を対象にしているため、必要最低限の情報でデバイスドライバが作成できるものと思われる。

本研究では、ハードウェア抽象化の無いハードウェアプラットフォームを対象としており、より詳細なハードウェア情報を用いてゲートウェイドライバの生成を行う。

DSL を用いた研究として、Voelter ら [37] は、組込み開発向けの拡張言語の研究を行っている。組込みソフトウェアに必要な機能を言語機能として提供することにより、効率よく組込みソフトウェアを開発することができる。

Hausmann ら [39] は、UML のアクティビティモデルを用いた System on Chip の高位合成を提案している。アクティビティモデルからは C 言語をベースとした高位合成言語である Handel-C を生成し、その後 RTL へと変換する。我々のソフトウェアからのアプローチと異なり、モデルからハードウェア自体を生成するアプローチであるが、抽象度の高いモデルからソフトウェア、ハードウェアの連携を行う点において共通している。

7.3 プロダクトライン開発とモデル駆動開発

ソフトウェアプロダクトライン開発においてフィーチャモデルから製品を導出する技術としては構成管理ツールの拡張が用いられることが多い。

例えばソフトウェアプロダクトライン開発向けの商用ツールとして、pure::variants[43] や Gears[44] が挙げられる。これらのツールは単なるビルド支援的な機能をベースに、さらに製品の構成を決めるための様々なディシジョンや導出のための複雑な操作が記述できる独自言語を持っている。

一方、フィーチャモデルからの製品導出に MDA を適用する研究もある。Muthing[40] は、OMG の MDA のに基いて、ソフトウェアのモデル中に可変点を定義し、フィーチャモデルからフィーチャ選択を行い可変点を確定することでモデルを完成させ、製品導出を行っている。また、Tawhid [41] は、UML と MARTE プロファイルを用いたソフトウェアプロダクトライン開発を提案している。MARTE プロファイルを用いることで、パフォーマンス分析を含めたシステム開発を行うことができる。Czarnecki ら [42] は、テンプレートを用いたフィーチャモデルから UML モデルへのマッピングを提案している。

このように、ソフトウェアプロダクトラインと MDA とを関連付けた研究はあるが、本研究ではフィーチャモデルの表している製品を導出するのではなく、ハードウェアとソフトウェアのそれぞれのフィーチャモデルから、それらの間のゲートウェイドライバの導出する点が新しいと言える。

製品系列を意識しながらハードウェアとのインタフェースとなるソフトウェアを生成する技術として AUTOSAR がある。AUTOSAR はこれらの複雑な車載組込みシステムのハードウェアとソフトウェアの統合的な開発を支援する標準規格で、ここではハードウェアモデルからのコード生成をサポートしているが、本研究のようなハードウェア構成に応じたゲートウェイドライバの生成は行っていない。AUTOSAR では、RTE (RunTime Environment) と呼ばれるミドルウェアレイヤが提供されており、このインタフェースに応じたハードウェアモデル、ゲートウェイドライバを設計することにより、ハードウェア、ソフトウェア間の整合性を保つ方式を用いている。

しかしながらハードウェアプラットフォームの標準化が難しい組込みシステムを想定しており、AUTOSARA のアプローチは適用が難しい。本研究のアプローチはそうした組込みシステムを対象とした新しい手法を提案するものである。

第 8 章

議論

8.1 貢献

本研究では，組込みシステム向けシステムプロダクトライン開発において導出されるハードウェアプロダクトに応じて必要となるゲートウェイドライバを自動的に導出する手法を提案した。

2層フィーチャモデルにより，ハードウェアの製品系列とソフトウェアの製品系列を独立に扱えるようにした。またハードウェアとソフトウェア間の依存関係の問題と，導出される組み合わせ数が膨大になる点について，段階的なフィーチャモデルの導出により，無効なソフトウェアフィーチャを排除し，組み合わせ数を削減することができた。

またハードウェアプロダクトとソフトウェアプロダクトのインタフェースとなるゲートウェイドライバを体系的に生成することができるようになり，システム製品群の開発がより容易になる。

一方，開発プロセスをモデル駆動のモデル変換として定義することにより，変換を自動化することができる点も上げられる。またハードウェア設計の情報をソフトウェア設計情報として取り入れることで，多様なハードウェアプロダクトに対応できる点がある。既に既存資産がある場合にはその取り込みが課題となる。本手法はモデルからのコード生成を行うため，既存資産を一度モデルへ取り入れる必要がある。既存資産のモデル化が難しい場合，差分開発で新規導入する部分のみをモデルで扱い，それ以外の部分は既存資産を利用するなどの対応が求められる。

本研究では、提案した手法をモデル駆動開発技術を用いて実装した、ハードウェア情報を含め、フィーチャモデルなども同一の技術を用いてメタモデルを定義することで、容易に複数の技術を組み合わせることができた。また、従来であればメタモデルレベルの変更が加わればそのモデルを扱っているすべての箇所に変更が必要となるが、EMF, GMF, Xtext の組合せでは環境のほとんどが自動生成であるため、メタモデルの作成を行えば、後は自動生成の途中段階で必要となる多少の情報をうまくコントロールできれば、新しいメタモデルに適した環境が生成される。こうした点はモデル駆動開発の利点であると考える。

8.2 適用性

本手法の適用範囲は、共通プラットフォームを持たない組込みシステムを対象としている。共通プラットフォームを持つ環境においても適用は可能であるが、ゲートウェイドライバが統一化されており自動生成のメリットが少ない点と、OSやミドルウェアなどの持つ機能を利用する枠組みがないことにより、本研究を適用する利点は少ない。

また、ゲートウェイモデルの導出に関する制約として、現在DMAなどのハードウェア機能に対するモデル化やゲートウェイドライバの導出は行えていない。加えてOSへの対応も行えていない。これは各OSのモデルやコードテンプレートを用意する必要があるため、作業量が膨大になってしまうためである。

システムプロダクトラインに関する制約として、現在の2層フィーチャモデルでは、ハードウェア・ソフトウェア間の制約として、機能レベルの制約しか設けていない。このため非機能の制約、例えばメモリや時間制約などは現状考慮出来ていない。

組込みシステムのデバイスドライバではしばしばパフォーマンスやタイミング等のチューニングが必要となることがあるが、モデルで表現できる範囲ではチューニングし切れない部分もある。このような場合、生成後のコードに対してチューニングを行う必要がある。

MARTEでは非機能特性のための様々な記述が可能であるため、ハードウェアプロダクトからソフトウェアフィーチャモデルに対して制約を掛ける際に、これら

の非機能特性を参照することで、非機能の制約を加えることが出来ると思われる。

導出されるゲートウェイモデルやドライバの性能については、記述したモデルや、用いるコードテンプレートの質による。コード生成の都合上、多少冗長なコードが入ってしまうこともあるが、現状では性能に大きく影響するほどの差異は見られていない。最低限までコード量を削減する必要がある場合は、生成されたコードに対してリファクタリングが必要となる。

8.3 再利用性

ハードウェア情報をモデル化することにより、ハードウェア情報自体の再利用ができるようになった。また、従来はゲートウェイドライバ開発は暗黙的で再利用することは難しかったが、ハードウェア情報からゲートウェイドライバまでの導出をモデル変換により明示的に定義できるようになった。これにより、導出のプロセス自体も再利用でき、資産化することが可能となった。

2層フィーチャモデルを用いたシステムプロダクトライン開発においては、ハードウェアプロダクトとソフトウェアプロダクトの間のインタフェースとなるゲートウェイドライバの体系だった導出が重要となる。開発者はハードウェア製品やソフトウェア製品のフィーチャ選択を行い効果的に製品導出したいが、両者のインタフェースは明示的に意識するものではなく、また問題で指摘したように一般にその開発は通常のソフトウェア開発とは異なった難しさがあるからである。

本研究ではこのゲートウェイドライバの自動的な導出方法を提案し、例題を用いてゲートウェイドライバを自動導出できることを実証した。本研究で扱ったようなデバイスの通信方式を定型的に使う限りは基本的に本提案で自動導出が可能と考える。

第 9 章

結論

9.1 結論

本研究では，小中規模の組込みシステムを対象に，ハードウェアプロダクトラインとソフトウェアプロダクトラインの両方を持つシステムプロダクトラインの開発手法を提案した．本提案の特徴は両プロダクトラインの可変性を捉える 2 層のフィーチャモデルを活用する点と，両プロダクトのインタフェースとなるゲートウェイドライバ部分を，ハードウェアプロダクトの情報に基づいて，モデル駆動によって自動導出する点である．提案にそった支援環境を実装し，その実現可能性の確認と評価を行った．

これにより中・小規模の組込みソフトウェアに特化した，抽象度の高いフィーチャモデルのレベルからソースコードまでの一貫したモデル駆動開発を実現できる．本研究を通していくつかの知見をえることが出来た．EMF ならびに GMF, Xtext を用いた開発はとても高い抽象度で実装を扱うことができ，ソフトウェアの多くは自動生成される．このため，モデル駆動の開発環境のような大規模なシステムであっても個人レベルで十分に開発を行うことが可能であり，本提案は小中規模の組込みシステムを開発する比較的小規模な開発組織においても十分適用性のある手法であると考えられる．

今後の課題として，今回の制約となっている DMA などのデバイスへの対応や，近年中規模組込みソフトウェアでも利用されるようになってきた OS への対応などが挙げられる．

近年デバイス技術の発達, IoTなどの進展の中, 組み込みシステムの重要性はますます高まってきている. そうした中, ハードウェアとソフトウェアからなるシステムの開発の効果的な支援は一層重要となっている. 本研究がそうした開発や研究へのひとつの貢献になることを願っている.

謝辞

まず何よりも博士前期課程，博士後期課程に渡り研究をご指導頂きました岸知二教授に深く感謝致します。何度も研究に行き詰まり論文も遅筆であった私を辛抱強くご指導頂き導いて頂きました。岸先生の元でなければ本論文まで辿りつけなかったと思います。本当にありがとうございました。

博士後期課程の研究室での生活や論文，研究の進め方について数々のご助言を頂きました Defago Xavier 先生に感謝致します。また青木利晃先生にも研究や研究室生活など様々にご助言，ご協力頂きましたありがとうございました。研究に関してご助言やご協力を頂きました野田夏子先生にも感謝致します。

九州大学に移ってからは，福田晃先生，鷓林尚靖先生始め，多くの先生方にご助言，ご協力頂きました。深く感謝いたします。JAISTでの生活を楽しく支えて頂いた研究室メンバーにも感謝致します。

最後にここまでの研究生生活を支え，見守ってくれた両親に感謝致します。皆様本当にありがとうございました。

参考文献

- [1] OSEK/VDX group, OSEK-VDX Portal, <http://www.osek-vdx.org/> (accessed 2014-08-24)
- [2] ITRON Project, ITRON Project Archive, <http://www.ert1.jp/ITRON/home-j.html> (accessed 2014-08-24)
- [3] FreeRTOS, FreeRTOS, <http://www.freertos.org/> (accessed 2014-08-24)
- [4] Pohl, K., Bockle, G. and van der Linden, F. Software Product Line Engineering . Foundations, Principles, and Techniques. Springer, Berlin, Heidelberg, New York, 2005.
- [5] Object Management Group: Model-Driven Architecture, www.omg.com/mda. (accessed 2014-08-24)
- [6] OMG. , Meta Object Facility (MOF) Version 2.4.1., <http://www.omg.org/spec/MOF/2.4.1/PDF> (accessed 2014-08-24)
- [7] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth, DSL Engineering: Desingning, Implementing, and Using Domain-Specific Languages, dslbook.org, (2013)
- [8] Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. : Addison-Wesley, (2004)
- [9] Itemis, Xtext, <http://www.eclipse.org/Xtext/> (accessed 2014-08-24)

- [10] 田中 明, 細合 晋太郎, テキスト型DSL 開発フレームワーク Xtext 入門, 技術評論社ベータパブリッシング, <http://www.beta-publish.com/books.html> (accessed 2014-08-24)
- [11] MetaCase, MetaEdit+ Domain-Specific Modeling (DSM) environment ,<http://www.metacase.com/products.html> (accessed 2014-08-24)
- [12] Eclipse org, Graphical Modeling Framework (GMF) ,<http://www.eclipse.org/modeling/gmf/> (accessed 2014-08-24)
- [13] Eclipse org, Graphiti, <http://www.eclipse.org/graphiti/> (accessed 2014-08-24)
- [14] Eclipse org, Sirius, <http://www.eclipse.org/sirius/> (accessed 2014-08-24)
- [15] OMG. ,UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), V1.1, <http://www.omg.org/spec/MARTE/1.1/PDF> (accessed 2014-08-24)
- [16] OMG. ,OMG Systems Modeling Language (OMG SysML), V1.2. , <http://www.omg.org/spec/SysML/1.2/PDF> (accessed 2014-08-24)
- [17] AUTOSAR. ,AUTOSAR., www.autosar.org (accessed 2014-08-24)
- [18] OMG. ,OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1, <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF> (accessed 2014-08-24)
- [19] Kang K. C., Cohen S. G., Hess J. A., Novak W. E., and Peterson A. S., "Feature-oriented domain analysis (FODA) feasibility study", Technical report, DTIC Document, 1990.
- [20] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, " FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering*, vol. 5, 1998, pp. 143 - 168.
- [21] eclipse.org., Eclipse, <http://www.eclipse.org/> (accessed 2014-08-24)

- [22] eclipse.org, Eclipse Modeling Framework, <http://www.eclipse.org/emf> (accessed 2014-08-24)
- [23] CEA., Papyrus UML, <http://www.eclipse.org/papyrus/> (accessed 2014-08-24)
- [24] Itemis, Xtext, <http://www.eclipse.org/Xtext/> (accessed 2014-08-24)
- [25] Eclipse org, Graphical Modeling Framework (GMF), <http://www.eclipse.org/modeling/gmf/> (accessed 2014-08-24)
- [26] Atmel, ATmega328P, <http://www.atmel.com/devices/ATMEGA328.aspx> (accessed 2014-08-24)
- [27] Kionix, KXP84, <http://www.kionix.com/accelerometers/kxss5>(accessed 2014-08-24)
- [28] Atmel, AT25010B., <http://www.atmel.com/devices/AT25010B.aspx> (accessed 2014-08-24)
- [29] Atmel, ATmega2560P, <http://www.atmel.com/devices/atmega2560.aspx> (accessed 2015-01-10)
- [30] InvenSense, MPU9150, <http://www.invensense.com/mems/gyro/mpu9150.html> (accessed 2015-01-10)
- [31] FTDI, FT232R, <http://www.ftdichip.com/Products/ICs/FT232R.htm>, (accessed 2015-01-10)
- [32] C. W. Krueger, “ New Methods in Software Product Line Development, ” in Proceedings of the International Software Product Line Conference (SPLC). IEEE Computer Society Press, 2006, pp. 95-102.
- [33] M. Rosenmuller and N. Siegmund, ”Automating the configuration of multi software product lines.”, In Fourth International Workshop on Variability Modelling of Software-intensive Systems, 2010.

- [34] Krzysztof Czarnecki, Simon Helsen, Ulrich Eisenecker "Staged Configuration Using Feature Models", LNCS Volume 3154, 2004, pp 266-283 2004
- [35] Jules White, Brian Dougherty, Douglas C. Schmidt, David Benavides, "Automated reasoning for multi-step feature model configuration problems". Proceeding SPLC '09 Proceedings of the 13th International Software Product Line Conference Pages 11-20, 2009
- [36] Hui Chen, Godet-Bar G., Rousseau F, Petrot F, "Me3D: A model-driven methodology expediting embedded device driver development," Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on , vol., no., pp.171,177, 24-27 May 2011
- [37] M Voelter, "Embedded Software Development with Projectional Language Workbenches", MoDELS, Lecture Notes in Computer Science, Vol.6395, Springer, pp.32-46, 2010.
- [38] S., Marlet, R. and Consel, C. Thibault, "Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation", IEEE Trans.Software Eng., Vol.25, No.3, pp.363-377, 1999.
- [39] T., Hausmann, J. H. and Engels, G. Schattkowsky, "Using UML Activities for System-on-Chip Design and Synthesis", MoDELS, Lecture Notes in Computer Science, Vol.4199, Springer, pp.737-752 2006.
- [40] Dirk Muthig, Colin Atkinson, "Model-Driven Product Line Architectures", SPLC 2002, LNCS Volume 2379, 2002, pp 110-129, 2002.
- [41] Tawhid, R.; Petriu, D.C., "Product Model Derivation by Model Transformation in Software Product Lines", Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on , vol., no., pp.72,79, 28-31 March 2011
- [42] K. Czarnecki, M. Antkiewicz, " Mapping Features to Models: A Template Approach Based on Superimposed Variants ", Proceedings of the 4th interna-

tional conference on Generative Programming and Component Engineering (GPCE), LNCS vol. 3676, pp. 422-437, Springer, 2005.

[43] Pure Systems, pure variants, <http://www.pure-systems.com/Home.142.0.html> (accessed 2014-08-24)

[44] BigLever, Gaers, <http://www.biglever.com/> (accessed 2014-08-24)

[45] CadSoft, Eagle, <http://www.cadsoftusa.com/> (accessed 2015-01-10)

本研究に関する発表論文

- [1] 細合 晋太郎, 岸 知二: ハードウェア情報を含めた MDA の提案と実装, 情報処理学会研究報告, 2007-EMB-005, No.52, pp. 3340 (2007).
- [2] 細合晋太郎, 岸知二, ”システム構成による制約を考慮した SPL の提案”, 情報処理学会, ソフトウェア・エンジニアリングシンポジウム 2008
- [3] 細合晋太郎, 岸知二, “ハードウェア情報のモデル化とモデル駆動技術による組込みソフトウェア開発への活用”, 情報処理学会, ソフトウェアエンジニアリングシンポジウム 2011, pp.1-8, 2011.
- [4] 細合晋太郎, 岸知二, “2 層フィーチャモデルを用いた開発手法の提案と実装”, 情報処理学会, 組込みシステム研究会, 2008- EMB-010, pp.29-36, 2008.
- [5] 細合晋太郎, 野田夏子, 岸知二, ”組み込みシステムプロダクトライン開発のためのドライバ生成方式の提案”, 情報処理学会 論文誌 (査読中)

付録

付録 A メタモデル定義

図 A.1 に本研究で実装したハードウェアメタモデル，ゲートウェイメタモデルのパッケージ構造を，図 A.2 に詳細なメタモデル定義，図 A.3 に詳細なメタモデルのリストを示す。

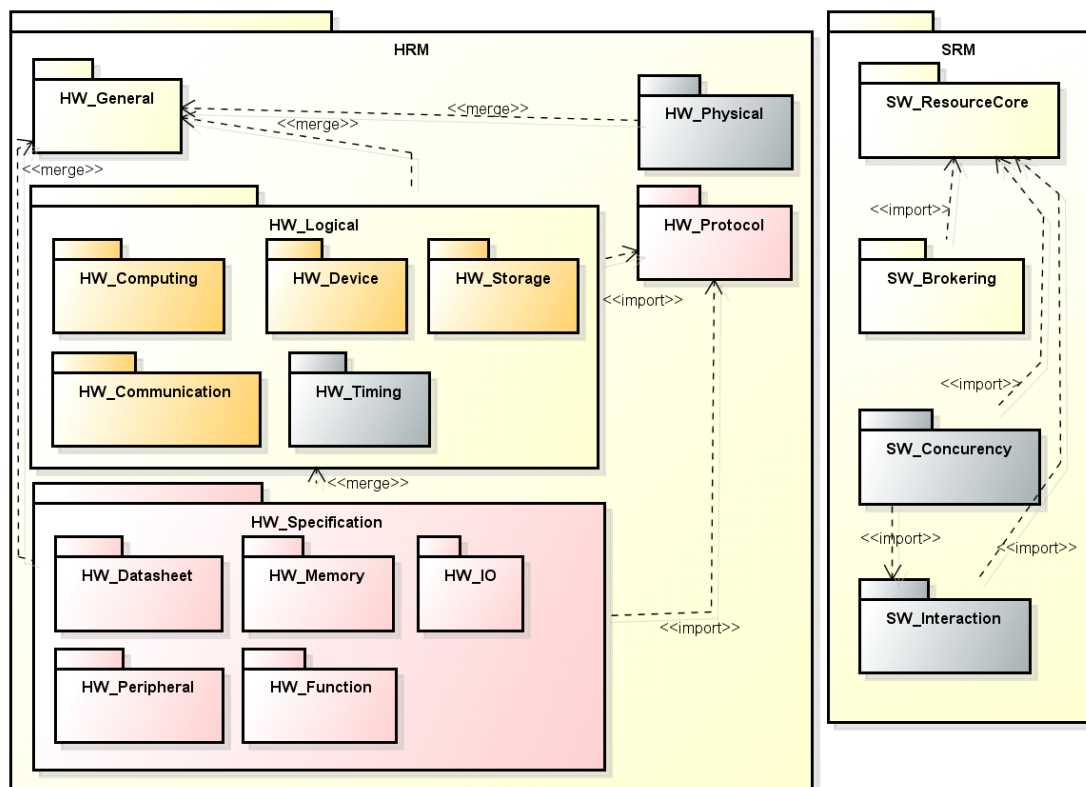


図 A.1: ハードウェアメタモデル，ゲートウェイメタモデルのパッケージ構造

付録 B 実装システムの構成

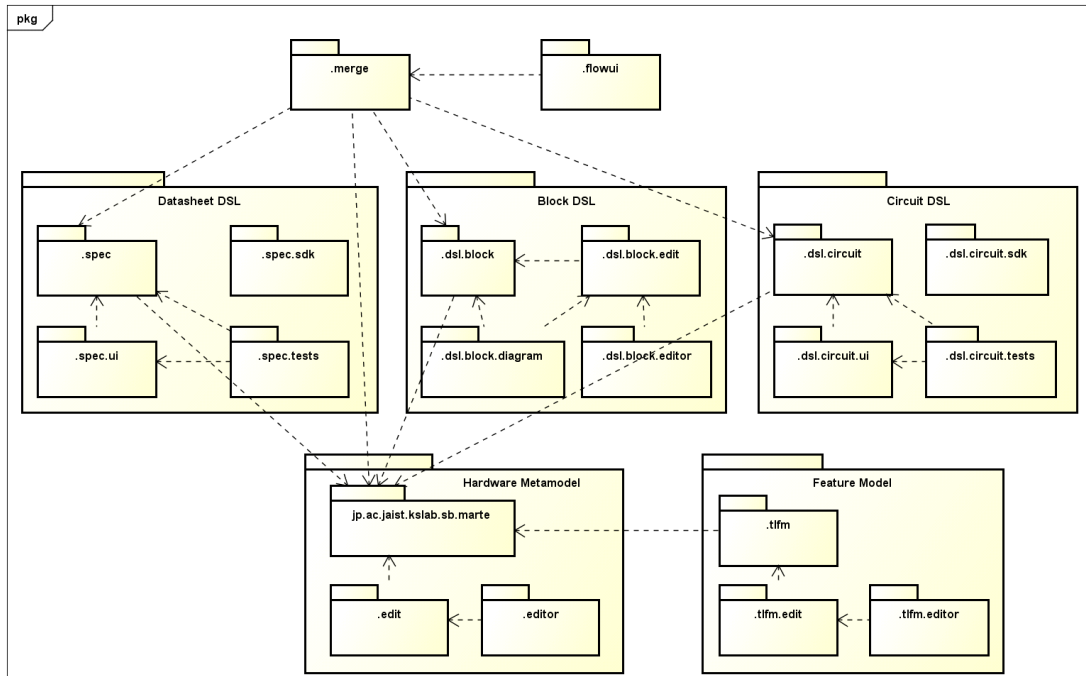


図 A.4: 実装システムの構成

付録 C DSL 定義

Code 1: ハードウェア部品 DSL の定義 (Xtext 形式)

```
grammar jp.ac.jaist.kslab.sb.marte.spec.SpecDsl with
org.eclipse.xtext.common.Terminals generate specDsl "http://www.ac
.jp/jaist/kslab/sb/marte/spec/SpecDsl"
import "http://www.eclipse.org/emf/2002/Ecore"
Model:
    Datasheet|DslProtocol;

Datasheet :
    ('revision' revision=PackageName)?
    'datasheet' name=ID
    '{'
        (imports += Import)*
        (components+=DslComponent)*
    '}' ;

Import:
    'import' importedNamespace = QualifiedNameWithWildcard;

DslProtocol:
    'protocol' name=QualifiedName '{'
        operations+=DslOperation*
    '}' ;

DslOperation:
    (type=DslType)? name=ID '(' (paramType+=DslType paramname+=
        ID (',' paramType+=DslType paramname+=ID)*)? ')' ;

DslComponent :
    DslMcu|DslDevice;

DslMcu :
    'mcu' name=ID '{'
        'pins_' pins+=DslPin (',' pins+=DslPin)* '}'
        ports+=DslPort*
        peripherals+=DslPeripheral*
        ('sfrs_'
```

```

        sfr+=DslRegister (','sfr+=DslRegister)*
    '})'?
    packages+=DslPackage*
'>';

DslDevice:
    'device' name=ID '{'
        'pins_{' pins+=DslPin (',' pins+=DslPin)* '}'
        ports+=DslPort*
        functions+=DslFunction*
        packages+=DslPackage*
        ('registers' '{'
            registers+=DslRegister (',' registers+=DslRegister
                )*
            '})'?
    '>';

/* common */
DslPort :
    'port' name=ID ('{'
        pins+=[DslPin] (',' pins+=[DslPin])*
    '})?';

DslPin :
    name=PinName;

/* memory */
DslRegister:
    address=HEX ':' name=ID ('{'(bits+=DslBit)? (',' (bits+=
        DslBit)?)*'})?';

DslBit:
    name=ID;

/* Packages */
DslPackage:
    'package' names+=ID (','names+=ID)*
    '{'
        pins+=DslPackagePin (',' pins+=DslPackagePin)*
    '>';

```

```

DslPackagePin:
    name=PinName '('pinNo=PinName')'('{
        altNames+=PinName (',' altNames+=PinName)*
    })?;

/* mcu specific */
DslPeripheral :
    'peripheral' name=ID 'implements' compliant+=[DslProtocol]
    '{
        operations+=DslOperationImpl*
    }';

DslOperationImpl :
    'impl' (rettype=DslType)? implOperation=[DslOperation]
    '('(params+=DslTypedParameter(','params+=DslTypedParameter
        )*)?)' '{
    expression+=DslPeripheralExpression*
    }';

DslPeripheralExpression:
    DslExpression;

DslRegisterRead :
    ref=XBoolType '=>' var=XBoolType;
    register=[DslRegister] "=>"(var=[DslVariable])?;

DslRegisterWrite:
    ref=XBoolType '<=' var=DslValue;

DslValue:
    HEX | BINARY | INT | XBoolType;

DslRefVariable:
    ref=[DslVariable];

/* Device Specific */
DslFunction :
    'function' (type=DslType)? name=ID '('(params+=
        DslTypedParameter(','
            params+=DslTypedParameter)*?)' '{
        expressions+=DslExpression*
    }';

```

```

    '}' ;

DslTypedParameter:
    paramType=DslType name=ID;

DslExpression:
    DslCallFunction | DslDefineVariable | DslReturn;

DslCallFunction:
    (variables=XBoolType "=")?function=[DslOperation |
    QualifiedName] '('(params+=XConditionalExpression (','
    params+=XConditionalExpression)*)?')';

DslDefineVariable:
    "var" name=ID ('=' val=XBoolType)?;

DslReturn:
    "return" val=XBoolType;

DslType :
    type=DslCType;

DslVariable:
    var=[DslDefineVariable];

/* basics */
Name:
    STRING | INT | ID;

QualifiedName:
    ID ('.' ID)*;

QualifiedNameWithWildcard:
    QualifiedName '.*'?;

PinName:
    Name;

PackageName:
    ('this'|ID) ('/' ID)*;

```

```

DslCType:
    'int' | 'char' | 'byte' | 'long' | 'float' | 'double' | 'string' | 'void'
    | 'bool';

terminal HEX returns EString:
    '0x'('A'..'F'|'0'..'9')+;

terminal BINARY returns EString:
    '0b'('0'..'1')+;

DslExpression:
    XPrimaryExpression | XBinaryExpression | XBlockExpression
    |
    DslCallFunction | DslDefineVariable | DslReturn |
    DslRegisterRead |
    DslRegisterWrite;

XBinaryExpression:
    left=DslVariable op=XBinaryOperator right=DslExpression;

XBinaryOperator:
    '+' | '-' | '=' | '!=' | '<' | '>' | '<<' | '>>' | '*' | '/' | '&' | '|';

XBlockExpression:
    '('expression=DslExpression)';

XPrimaryExpression:
    XIfExpression | XWhileExpression | XForExpression |
    XDoWhileExpression;

XConditionalExpression:
    XConditionalUnaryExpression | XConditionalBlockExpression |
    XBoolExpression | XBoolType | XLiteral;

XConditionalBlockExpression:
    '('expression=XConditionalExpression)';

XConditionalUnaryExpression:
    '!expression=XConditionalExpression;

XBoolExpression:

```

```

        left=XBoolType op=XBoolOperation right=
            XConditionalExpression;

XBoolOperation:
    '|'| '&&' | '==' | '!=' | '>' | '<' | '>=' | '<=' | '&' | '|';

XBoolType:
    XBitAccess | XRegisterAccess | XVariableAccess |
        XParamVariableAccess;

XLiteral:
    BINARY | HEX | INT;

XBitAccess:
    '#' bit=[DslBit | QualifiedName];

XRegisterAccess:
    '$' reg=[DslRegister];

XVariableAccess:
    var=[DslDefineVariable];

XParamVariableAccess:
    '~' var=[DslTypedParameter];

XIfExpression:
    'if' '(' if=XConditionalExpression ')' '{'
        then=DslExpression
    '}' ('else' '{' else=DslExpression '}')?;

XWhileExpression:
    'while' '(' predicate=XConditionalExpression ')' '{'
        (body=DslExpression)?
    '}'

XForExpression:
    'for' '(' init=DslExpression ';' predicate=
        XConditionalExpression ';'
    loop=DslExpression ')' '{'
        (body=DslExpression)?
    '}'

XDoWhileExpression:

```

```

    'do' '{'
        body=DslExpression
    '}' 'while' '(' predicate=XConditionalExpression ')';

```

Code 2: 回路図 DSL の定義 (Xtext 形式)

```

grammar jp.ac.jaist.kslab.sb.marte.dsl.circuit.CircuitDsl with org
    .eclipse.xtext.common.Terminals

import 'http://MARTE/MARTE_DesignModel/HRM/HwDiagram.ecore'
import 'http://MARTE/MARTE_DesignModel/HRM/HwSpecification/
    HwPackage.ecore'

HwCircuitDiagram returns HwCircuitDiagram:
{HwCircuitDiagram}
    'net' '{'
        wires+=HwWire(',' wires+=HwWire)*
    '}'
    components+=HwComponents*;

HwComponents returns HwPackage:
    'component' name=ID '{'
        pins+=HwPackagePin(',' pins+=HwPackagePin)*
    '}' ;

HwPackagePin returns HwPackagePin:
    name=ID ':' wire+=[HwWire];

HwWire returns HwWire:
    name=ID;

```

付録 D 変換定義ソースコード

Code 3: DSL から各ハードウェアモデルへの変換定義 (Xtend 形式)

```

package jp.ac.jaist.kslab.sb.marte.merge.generator

import MARTE.MARTE_DesignModel.HRM.HwDiagram.HwDiagramFactory
import MARTE.MARTE_DesignModel.HRM.HwDiagram.HwHRMDiagram

```

```

import MARTE.MARTE_DesignModel.HRM.HwLogical.HwCommunication.
    HwCommunicationFactory
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwCommunication.
    HwPort
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwComputing.
    HwComputingFactory
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwComputing.HwMCU
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwDevice.HwDevice
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwDevice.
    HwDeviceFactory
import MARTE.MARTE_DesignModel.HRM.HwProtocol.HwProtocolFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.
    HwDeviceFunction.HwDeviceFunctionFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwIO.
    HwIOFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwIO.HwPin
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwPackage.
    HwPackage
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwPackage.
    HwPackageFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwPackage.
    HwPackagePin
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwPeripheral.
    HwPeripheralFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwRegister.
    HwRegisterFactory
import java.util.ArrayList
import java.util.HashMap
import java.util.Map
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.Datasheet
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslCallFunction
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslDefineVariable
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslDevice
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslExpression
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslFunction
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslMcu
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslOperation
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslOperationImpl
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslPackage
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslPackagePin
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslPeripheral

```



```

import jp.ac.jaist.kslab.sb.marte.spec.specDsl.
    DslPeripheralExpression
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslPin
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslPort
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslProtocol
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslRegister
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslRegisterRead
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslRegisterWrite
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslType
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.DslTypedParameter
import org.eclipse.emf.common.util.EList
import org.eclipse.emf.ecore.EObject
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.emf.mwe.core.WorkflowContext
import org.eclipse.emf.mwe.core.issues.Issues
import org.eclipse.emf.mwe.core.lib.WorkflowComponentWithModelSlot
import org.eclipse.emf.mwe.core.monitor.ProgressMonitor
import org.eclipse.uml2.uml.DataType
import org.eclipse.uml2.uml.UMLFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwRegister.
    HwRegister
import jp.ac.jaist.kslab.sb.marte.spec.specDsl.XRegisterAccess

class DSLTransformer extends WorkflowComponentWithModelSlot {
// @Inject extension DslBehaviorTransformer

    // workflow params
    private String input
    private String output

    def setInput(String inputSlot){
        this.input = inputSlot
    }

    def setOutput(String outputSlot){
        this.output = outputSlot
    }

    // transform params
    private Map<HwPort, ArrayList<String>> unresolvePortPins = new
        HashMap()

```

```

private Map<String, DataType> types
private Map<String, HwRegister> registers;

override protected invokeInternal(WorkflowContext ctx,
    ProgressMonitor monitor, Issues issues) {
    var dslmodels = ctx.get(input) as ArrayList<Resource>
    var ArrayList<EObject> results = new ArrayList<EObject>()

    var HwHRMDiagram result
    for(r: dslmodels){
        types = new HashMap()
        for(e : r.allContents.toIterable){
            switch(e){
                Datasheet: {
                    result = createDatasheet(e)
                    result.pinReconnect()
                }
                DslProtocol : result = createProtocolDatasheet(e)
            }
        }
        result.datatypes.addAll(types.values)
        results.add(result)
    }
    ctx.set(output, results)
}

def create r : HwDiagramFactory::eINSTANCE.createHwHRMDiagram
    createProtocolDatasheet(DslProtocol d){
    r.name = d.name
    r.protocols.add(createProtocol(d))
}

def create r : HwDiagramFactory::eINSTANCE.createHwHRMDiagram
    createDatasheet(Datasheet d){
    for(component : d.components){
        registers = new HashMap()
        switch(component){
            DslDevice : r.components.add(createComponent(component))
            DslMcu : r.components.add(createComponent(component))
        }
    }
}

```

```

for(imp : d.imports){
  if(!imp.importedNamespace.contains("*")){
    r.protocols.add(createProtocol(imp.importedNamespace))
    println("add_protocol"+imp.importedNamespace)
  }
}

def create r : HwComputingFactory::eINSTANCE.createHwMCU
  createComponent(DslMcu mcu){
    r.name = mcu.name
    for(pin : mcu.pins){
      r.pins.add(createPin(pin))
    }
    for(port : mcu.ports){
      r.ports.add(createPort(port))
    }
    for(pkg : mcu.packages){
      r.packages.add(createPackage(pkg))
    }
    for(reg : mcu.sfr){
      r.sfr.add(createRegister(reg))
    }
    for(peripheral : mcu.peripherals){
      r.peripherals.add(createPeripheral(peripheral))
    }
}

def create r : HwDeviceFactory::eINSTANCE.createHwDevice
  createComponent(DslDevice dev){
    r.name = dev.name
    for(pin : dev.pins){
      r.pins.add(createPin(pin))
    }
    for(port : dev.ports){
      r.ports.add(createPort(port))
    }
    for(pkg : dev.packages){
      r.packages.add(createPackage(pkg))
    }
    for(reg : dev.registers){

```

```

    r.registers.add(createRegister(reg))
  }
  for(func : dev.functions){
    var devfunc = createFunction(func)
    var devact = createActivity(func)
    devfunc.methods.add(devact)
    devact.specification = devfunc
    r.operations.add(devfunc)
    r.activities.add(devact)
  }
}

def create r : HwProtocolFactory::eINSTANCE.createHwProtocol
  createProtocol(DslProtocol protocol){
  r.name = protocol.name
  for(op : protocol.operations){
    r.operations.add(createOperation(op))
  }
}

def create r : HwProtocolFactory::eINSTANCE.createHwProtocol
  createProtocol(String name){
  r.name = name
}

// for common elements
//Pin
def create r : HwIOFactory::eINSTANCE.createHwPin createPin(
  DslPin pin){
  r.name = pin.name
  //pkg, のとはで. portpinpinreconnect
}
//Port
def create r : HwCommunicationFactory::eINSTANCE.createHwPort
  createPort(DslPort port){
  r.name = port.name
  var pins = new ArrayList<String>()
  for(p : port.pins){
    pins.add(p.name)
  }
  unresolvePortPins.put(r, pins)
}

```

```

    }
//Package
def create r : HwPackageFactory::eINSTANCE.createHwPackage
    createPackage(DslPackage pkg){
        r.name = pkg.names.head
        r.packageType = pkg.names.head

        for(pin : pkg.pins){
            r.pins.add(createPkgPin(pin))
        }
    }
//PkgPin
def create r : HwPackageFactory::eINSTANCE.createHwPackagePin
    createPkgPin(DslPackagePin pin){
        r.name = pin.name
        r.pinNo = pin.pinNo
        for(alt : pin.altNames){
            r.altNames.add(alt)
        }
    }
//Register
def create r : HwRegisterFactory::eINSTANCE.createHwRegister
    createRegister(DslRegister register){
        r.name = register.name
        r.address = register.address
        registers.put(register.name, r)
    }

// for mcu elements
def create r : HwDeviceFactory::eINSTANCE.createHwPeripheral
    createPeripheral(DslPeripheral peripheral){
        r.name = peripheral.name
        for(op : peripheral.operations){
            var opimpl = createOperationImpl(op)
            var actimpl = createPeripheralActivities(op)
            opimpl.methods.add(actimpl)
            actimpl.specification = opimpl
            r.operationimpls.add(opimpl)
            r.activities.add(actimpl)
        }
    }
}

```

```

//OperationImpl
def create r : HwPeripheralFactory::eINSTANCE.
    createOperationImpl createOperationImpl(DslOperationImpl op){
    r.name = op.implOperation.name
    r.type = getType(op.rettype)

    for(param : op.params){
//        r.parameterableElements.add(createParameter(param))
    }
}

def create r : HwPeripheralFactory::eINSTANCE.
    createPeripheralActivity createPeripheralActivities(
    DslOperationImpl op){
    r.name = op.implOperation.name

    for(param : op.params){
        r.ownedNodes.add(createPin(param))
    }
    r.ownedNodes.add(UMLFactory::eINSTANCE.createInitialNode)
    for(exp : op.expression){
        r.ownedNodes.add(createAction(exp))
    }
    r.ownedNodes.add(UMLFactory::eINSTANCE.createActivityFinalNode
    )
    if (op.rettype!=null){
        var result = UMLFactory::eINSTANCE.createOutputPin()
        result.name = op.rettype.type
        r.ownedNodes.add(result)
    }
}

def createAction(DslPeripheralExpression exp){
    switch(exp){
        DslRegisterRead:createAction(exp)
        DslRegisterWrite:createAction(exp)
        default : createActionAnyOther(exp)
    }
}
}

```

```

def create r : HwPeripheralFactory::eINSTANCE.
  createReadRegisterAction() createAction(DslRegisterRead exp){
var ref = exp.ref
switch(ref){
  XRegisterAccess:{
    r.name = ref.reg.name
    r.register = registers.get(ref.reg.name)
  }
}
}

def create r : HwPeripheralFactory::eINSTANCE.
  createWriteRegisterAction() createAction(DslRegisterWrite exp
){
var ref = exp.ref
switch(ref){
  XRegisterAccess:{
    r.name = ref.reg.name
    r.register = registers.get(ref.reg.name);
  }
}
}

def create r : UMLFactory::eINSTANCE.
  createStructuredActivityNode createActionAnyOther(
  DslPeripheralExpression exp){
  r.name = exp.toString
}

// for device elements
def create r : HwDeviceFunctionFactory::eINSTANCE.
  createHwDeviceFunction createFunction(DslFunction function){
  r.name = function.name
  r.setIsAbstract(false)
  r.type = getType(function.type)
}

def create r : UMLFactory::eINSTANCE.createOperation
  createOperation(DslOperation operation){
  r.name = operation.name
  r.setIsAbstract(true)
  r.type = getType(operation.type)
}

```

```

    r.ownedParameters
}

def create r : UMLFactory::eINSTANCE.createActivity
  createActivity(DslFunction func){
    r.name = func.name
    // input pins from parameter
    for(param : func.params){
      r.nodes.add(createPin(param))
    }
    r.nodes.add(UMLFactory::eINSTANCE.createInitialNode)
    for(exp : func.expressions){
      r.nodes.add(createAction(exp))
    }
    r.nodes.add(UMLFactory::eINSTANCE.createActivityFinalNode)

    if (func.type!=null){
      var result = UMLFactory::eINSTANCE.createOutputPin()
      result.name = func.type.type
      result.type = getType(func.type)
      r.nodes.add(result)
    }
  }

def create r : UMLFactory::eINSTANCE.createInputPin createPin(
  DslTypedParameter param){
  r.name = param.name
  r.type = getType(param.paramType)
}

def createAction(DslExpression exp){
  switch(exp){
    DslCallFunction:createAction(exp)
    DslDefineVariable:createAction(exp)
    default : createActionAnyOther(exp)
  }
}

def create r : UMLFactory::eINSTANCE.createCallOperationAction
  createAction(DslCallFunction exp){
    r.name = exp.function.name
  }
}

```



```

//Task
def create r : UMLFactory::eINSTANCE.createCallOperationAction
  createAction(DslDefineVariable exp){
  r.name = exp.name
}

def getType(DslType t){
  if(t!=null){
    if(types.containsKey(t.type)){
      return types.get(t.type)
    }else{
      var type = createType(t.type)
      types.put(t.type, type)
      return type
    }
  }else{
    var v = createType("void")
    types.put("void", v)
  }
}

def create r : UMLFactory::eINSTANCE.createDataType createType(
  String name){
  r.name = name
}

def pinReconnect(HwHRMDiagram d){
  for(comp : d.components){
    var EList<HwPin> pins
    var EList<HwPackagePin> pkgpins
    var HwPackage pkg
    var EList<HwPort> ports
    switch (comp){
      HwMCU:{
        pins = comp.pins
        pkg = comp.packages.head
        pkgpins = pkg.pins
        ports = comp.ports
      }
      HwDevice:{

```



```

import org.eclipse.emf.mwe.core.lib.AbstractWorkflowComponent
import org.eclipse.emf.mwe.core.monitor.ProgressMonitor
import org.eclipse.uml2.uml.Activity
import org.eclipse.uml2.uml.CallOperationAction
import org.eclipse.uml2.uml.DataType
import org.eclipse.uml2.uml.Operation
import java.util.List
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwPackage.
    HwWire
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwCommunication.
    HwCommunicationFactory
import MARTE.MARTE_DesignModel.HRM.HwGeneral.HwResource
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwIO.HwPin
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwPackage.
    HwPackage
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwCommunication.
    HwPort
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwIO.
    HwIOFactory
import MARTE.MARTE_DesignModel.HRM.HwProtocol.HwProtocolFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwIO.HwLine
import blockdiagram.MCU
import blockdiagram.Device
import org.eclipse.emf.mwe.core.WorkflowInterruptedException
import blockdiagram.Connection
import MARTE.MARTE_DesignModel.HRM.HwDiagram.HwDiagramFactory
import MARTE.MARTE_DesignModel.HRM.HwSpecification.HwPackage.
    HwPackagePin

class HardwareModelMerger extends AbstractWorkflowComponent {
    // slots
    String circuitSlot
    String blockSlot
    String specSlot
    String output

    //inter
    Map<String, HwMCU> mcus
    Map<String, HwDevice> devices
    Map<String, HwProtocol> protocols
    Map<String, DataType> datatypes

```

```

HwCircuitDiagram circuitdiagram
Diagram blockdiagram
HwHRMDiagram hardwaremodel

/**
 *
 */
override protected invokeInternal(WorkflowContext ctx,
    ProgressMonitor monitor, Issues issues) {
    println("loading_models...")
    mcus = new HashMap<String, HwMCU>()
    devices = new HashMap<String, HwDevice>()
    protocols = new HashMap()
    datatypes = new HashMap()

    var model = ctx.get(specSlot) as ArrayList<HwHRMDiagram>
    for(datasheet : model){
        for(protocol : datasheet.protocols as EList<HwProtocol>){
            if(protocol.operations.size!=0){
                protocols.put(protocol.name, protocol)
            }
        }
        for(type : datasheet.datatypes as EList<DataType>){
            datatypes.put(type.name, type)
        }
    }
    for(datasheet : model){
        for(com : datasheet.components){
            switch com {
                HwMCU : {
                    mcus.put(com.name, com)
                    println("puts_mcu_"+com.name)
                }
                HwDevice :{
                    devices.put(com.name, com)
                    for(act : com.activities as EList<Activity>){
                        for(action : act.nodes){
                            switch(action){
                                CallOperationAction:{
                                    var op = findOperation(action)

```

```

        if(op!=null) action.operation=op
    }
}
}
}
println("puts_dev"+com.name)
}
}
}

blockdiagram = ctx.get(blockSlot) as Diagram
println("merging_block_model")
hardwaremodel = createPortlevelHRM(blockdiagram)

println("merging_circuit_model")
circuitdiagram = getCircuitModel(ctx.get(circuitSlot) as
    ArrayList<Resource>)
if (circuitdiagram==null) {
    issues.addWarning("CIRCUIT_IS_NULL")
}

createDetailedHRM(hardwaremodel, circuitdiagram)
hardwaremodel.datatypes.addAll(datatypes.values)
ctx.set(output, hardwaremodel)
}

def findOperation(CallOperationAction act){
    for(p : protocols.values){
        var obj = p.operations.findFirst(Operation op|op.name==act.
            name) as Operation
        if(obj!=null){
            println("find:_"+obj.name)
            return obj
        }
    }
    return null
}

override checkConfiguration(Issues arg0) {
}

```

```

def HwCircuitDiagram getCircuitModel(ArrayList<Resource>
resources){
for(r : resources){
for(obj : r.allContents.toIterable){
switch obj {
HwCircuitDiagram : return obj
}
}
}
return null
// throw exception?
}

def create r : HwDiagramFactory::eINSTANCE.createHwHRMDiagram
createPortlevelHRM(Diagram block){
for(com : block.com){
switch(com){
MCU : {
var mcu = mcus.get(com.name)
if(mcu!=null){
r.components.add(mcu)
}else{
println("not found: "+com.name)
}
}
Device : {
var dev = devices.get(com.name)
if(dev!=null){
r.components.add(dev)
}else{
println("not found: "+com.name)
}
}
}
}
// connect port to port
return r
}

for(component : block.com as EList<blockdiagram.HwComponent>){
for(con : component.conection as EList<Connection>){

```

```

switch(component){
    MCU : {
        var port1 = mcus.get(component.name).ports.getPort(con
            .name)
        var port2 = devices.get(con.ref.name).ports.getPort(
            con.name)
        if(port1==null || port2==null){
            throw new WorkflowInterruptedException()
        }
        r.connections.add(createConnection(port1, port2, con))
    }
    Device : {
        var port1 = mcus.get(con.ref.name).ports.getPort(con.
            name)
        var port2 = devices.get(component.name).ports.getPort(
            con.name)
        if(port1==null || port2==null){
            throw new WorkflowInterruptedException()
        }
        r.connections.add(createConnection(port1, port2, con))
    }
}
}
}
}

def createDetailedHRM(HwHRMDiagram diag, HwCircuitDiagram
    circuit){
    var List<HwWire> wires = new ArrayList()
    var List<HwLine> lines = new ArrayList()
    for(w : circuit.wires as EList<HwWire>){
        wires.add(w)
        lines.add(createLine(w))
    }
    diag.connections.addAll(wires)
    diag.connections.addAll(lines)

    for(circuitComponent : circuit.components as EList<HwPackage>)
    {
        var pos = circuitComponent.name.indexOf("_")
        var compName = circuitComponent.name.substring(0,pos)

```

```

var pkgName = circuitComponent.name.substring(pos+1)
println(compName+",␣"+pkgName)
var component = getComponent(compName)
for(circuitPin :circuitComponent.pins as EList<HwPackagePin
    >){
    var pkgpin = getPkgPin(component, pkgName, circuitPin.name
)
    var pin = getPin(component, circuitPin.name)
    for(circuitWire : circuitPin.wire as EList<HwWire>){
        println(pkgpin)
        println(pin)
        pkgpin.wire.add(wires.findFirst(w | w.name==circuitWire.
            name))
        pin.lines.add(lines.findFirst(l | l.name==circuitWire.
            name))

    }
}
diag.protocols.addAll(protocols.values)
}

def create r : HwCommunicationFactory::eINSTANCE.
    createHwConnection createConnection(HwPort p1, HwPort p2,
    Connection blockConnection){
    r.name = blockConnection.name
    r.protocols.add(getProtocol(blockConnection.name))

    p1.connectedTo.add(r)
    p2.connectedTo.add(r)
}

def create r : HwIOFactory::eINSTANCE.createHwLine createLine(
    HwWire wire){
    r.name = wire.name
}

def create r : HwProtocolFactory::eINSTANCE.createHwProtocol
    createProtocol(String name){
    r.name = name
}

```



```

def getComponent(String name){
    var all = new HashMap<String, HwResource>()
    all.putAll(mcus)
    all.putAll(devices)
    all.get(name)
}

def getProtocol(String name){
    if (protocols.containsKey(name)){
        return protocols.get(name)
    }else{
//        var p = createProtocol(name)
//        protocols.put(name, p)
        return null
    }
}

def getPort(EList<HwPort> ports, String name){
    ports.findFirst(HwPort p | p.name==name)
}

def getPkgPin(HwResource res, String pkgName, String pinName){
    var EList<HwPackage> pkgs
    switch(res){
        HwMCU : pkgs = res.packages
        HwDevice : pkgs = res.packages
        default : return null
    }
    pkgs.findFirst(HwPackage p | p.name==pkgName).pins.findFirst(
        HwPackagePin pin | pin.name==pinName)
}

def getPin(HwResource res, String pinname){
    var EList<HwPin> pins
    switch(res){
        HwMCU: pins = res.pins
        HwDevice : pins = res.pins
    }
    pins.findFirst(p | p.name==pinname)
}

```

```

// setter, getter, others

def void setSpec(String specSlot){
    this.specSlot = specSlot
}
def void setCircuit(String circuitSlot){
    this.circuitSlot = circuitSlot
}
def void setBlock(String blockSlot){
    this.blockSlot = blockSlot
}
def void setOutput(String output) {
    this.output = output
}
}
}

```

Code 5: ハードウェアモデルからゲートウェイモデルへの変換定義 (Xtend形式)

```

package jp.ac.jaist.kslab.sb.marte.merge.generator

import MARTE.MARTE_DesignModel.HRM.HwDiagram.HwDiagramFactory
import MARTE.MARTE_DesignModel.HRM.HwDiagram.HwHRMDiagram
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwComputing.HwMCU
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwDevice.HwDevice
import MARTE.MARTE_DesignModel.HRM.HwLogical.HwDevice.HwPeripheral
import MARTE.MARTE_DesignModel.SRM.SW_Brokering.DeviceBroker
import MARTE.MARTE_DesignModel.SRM.SW_Brokering.
    SW_BrokeringFactory
import java.util.HashSet
import java.util.Set
import org.eclipse.emf.common.util.EList
import org.eclipse.emf.ecore.util.EcoreUtil
import org.eclipse.emf.mwe.core.WorkflowContext
import org.eclipse.emf.mwe.core.issues.Issues
import org.eclipse.emf.mwe.core.lib.AbstractWorkflowComponent
import org.eclipse.emf.mwe.core.monitor.ProgressMonitor
import org.eclipse.uml2.uml.Activity
import org.eclipse.uml2.uml.CallOperationAction
import org.eclipse.uml2.uml.Operation

class HRMtoSRMTransformer extends AbstractWorkflowComponent {

```

```

private String hardwareSlot
private String gatewaySlot

private DeviceBroker mcu
private Set<DeviceBroker> devs = new HashSet()

override protected invokeInternal(WorkflowContext ctx,
    ProgressMonitor monitor, Issues issues) {
    println("HRM□to□SRM□transforming")
    var model = ctx.get(hardwareSlot) as HwHRMDiagram
    var srmdiagram = createSRMDiagram
    for(com : model.components){
        switch(com){
            HwMCU:{
                var broker = createSRMComponent(com)
                srmdiagram.devices.add(broker)
                mcu =broker
            }
            HwDevice:{
                var broker = createSRMComponent(com)
                srmdiagram.devices.add(broker)
                devs.add(broker)
            }
        }
    }

    callReconnect()

    ctx.set(gatewaySlot, srmdiagram)
}

def create r: HwDiagramFactory::eINSTANCE.createSRMDiagram()
    createSRMDiagram(){

def create r: SW_BrokeringFactory::eINSTANCE.createDeviceBroker
    () createSRMComponent(HwMCU mcu){
    r.name = mcu.name
//    r.devices.add(mcu)
    for(peripheral : mcu.peripherals as EList<HwPeripheral>){
        r.operations.addAll(EcoreUtil::copyAll(peripheral.
operationimpls))

```

```

        r.activities.addAll(EcoreUtil::copyAll(peripheral.activities
        ))
    }
    for(op:r.operations as EList<Operation>){
        var act = r.activities.findFirst(Activity a | a.name==op.
name)
        act.specification=op
        op.methods.add(act)
    }
}

def create r: SW_BrokeringFactory::eINSTANCE.createDeviceBroker
() createSRMComponent(HwDevice dev){
    r.name = dev.name
//    r.devices.add(dev)
    r.operations.addAll(EcoreUtil::copyAll(dev.operations))
    r.activities.addAll(EcoreUtil::copyAll(dev.activities))
    for(op:r.operations as EList<Operation>){
        var act = r.activities.findFirst(Activity a | a.name==op.
name)
        act.specification=op
        op.methods.add(act)
    }
}

def callReconnect(){
    for(d : devs){
        for(act : d.activities as EList<Activity>){
            for(call : act.nodes){
                switch(call){
                    CallOperationAction : {
                        call.operation = mcu.operations.findFirst(Operation
o | o.name==call.operation.name)
                    }
                }
            }
        }
    }
}
}

```

```

// setter & check
def setHardwareModel(String hardwareSlot){
    this.hardwareSlot = hardwareSlot
}
def setGatewayModel(String gatewaySlot){
    this.gatewaySlot = gatewaySlot
}

override checkConfiguration(Issues issues) {
    if(hardwareSlot==null){
        issues.addError("Hardware Slot is empty")
    }
    if(gatewaySlot==null){
        issues.addError("Gateway Slot is empty")
    }
}
}
}

```

Code 6: ゲートウェイモデルからのコード生成定義 (Xtend 形式)

```

package jp.ac.jaist.kslab.sb.marte.merge.generator

import java.util.HashMap
import java.util.Map
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.uml2.uml.Activity
import org.eclipse.uml2.uml.ActivityNode
import org.eclipse.uml2.uml.Class
import org.eclipse.uml2.uml.Operation
import org.eclipse.uml2.uml.ParameterableElement
import org.eclipse.xtext.generator.IFileSystemAccess
import org.eclipse.xtext.generator.IGenerator
import org.eclipse.uml2.uml.Type
import org.eclipse.uml2.uml.CallOperationAction
import MARTE.MARTE_DesignModel.SRM.SW_Brokering.DeviceBroker

class UMLtoCodeGenerator implements IGenerator {
    Map<String, Activity> activities = new HashMap()
    override doGenerate(Resource input, IFileSystemAccess fsa) {
        for(r : input.allContents.toIterable){
            switch(r){
                Activity:{

```

```

        // nop
    }
    org.eclipse.uml2.uml.Class : {
        fsa.generateFile(r.name.toLowerCase+".h", generateHeader
            (r))
        fsa.generateFile(r.name.toLowerCase+".c", generateSource
            (r))
    }
    default:{
    }
}
}
}

def generateHeader(Class clazz)'''
    ##ifndef__<<clazz.name.toUpperCase>>_H
    ##define__<<clazz.name.toUpperCase>>_H
    ##<<FOR op :_<<clazz.ownedOperations>>
    ##<<generateSignature(op)>>
    ##<<ENDFOR>>
    ##endif
    '''
    def addActivities(Class clazz){
        activities.clear
        for(act : clazz.ownedBehaviors){
            activities.put(act.name, act as Activity)
        }
    }

    def generateSignature(Operation op)'''
    ##<<getTypeName(op.type)>>_<<op.name>>(<<for(p :_<<op.
        parameterableElements>>{generateParam(p)}>>);
    '''

    def generateParam(ParameterableElement elem)'''

    '''

    def generateSource(Class clazz)'''
    ##<<addActivities(clazz)>>
    ##include"<<clazz.name.toLowerCase>>.h"

```

```

    <<FOR inc : clazz.ownedAttributes>>
    #include "<<inc.name.toLowerCase>>.h"
    <<ENDFOR>>

    <<FOR op : clazz.ownedOperations>>
        <<generateMethod(op)>>
    <<ENDFOR>>
'''

    def generateMethod(Operation op)'''
    <<getTypeName(op.type)>> <<op.name>>(){
    <<FOR node : activities.get(op.name).ownedNodes>>
    <<genBody(node)>>
    <<ENDFOR>>
    }
    '''

    def genParam(ParameterableElement p)'''
    '''

    def dispatch genBody(ActivityNode a)'''
    '''

    def dispatch genBody(CallOperationAction call)'''
    <<call.name>>();
    '''

    def getTypeName(Type t){
        if(t==null || t.name==null){
            return "void"
        }else if(t.name.equals("byte")){
            return "unsigned_char"
        }else{
            return t.name
        }
    }
}

```