G-21

IoT フレームワークにおけるデバイス用プラグインのモデル駆動開発手法

Model-driven Development of Device Plugins for IoT Frameworks

金岡 優奈† 満田 成紀‡ 福安 直樹‡ 松延 拓生‡ 鯵坂 恒夫‡
Yuna Knaoka Naruki Mitsuda Naoki Fukuyasu Takuo Matsunobe Tsuneo Ajisaka

1. はじめに

コンピュータの発展により様々な分野において機械の自動化やネットワーク化が進み，IoT(Internet of Things)という仕組みが導き出された．IoT とは，様々なモノがインターネットに接続され，情報交換することで相互に制御する仕組みである．多様な機器がインターネットに繋がることで，システムが以前より複雑になっている．そのため，以前から懸念され続けられているシステムの大規模化がより一層問題視されている．これらの課題によって，ソフトウェア工学によるプロセス管理や要求分析技術， 効率的な開発手法・方法論などを開発現場に導入することが求められています．手法の一例としてモデル駆動開発が挙げられる．モデル駆動開発はシステムに要求される機能を様々な側面からモデルによって表し，モデルの情報を変換することでプログラムを生成できる手法である．また，機器の多様化に伴い，デバイスの接続方法も多様化されており，通信方式に対応させるのが困難になっている．そこで導入されている一例が“GotAPI”というフレームワークである．このフレームワークの一部を構成するプラグインというソフトウェアは，デバイスごとの通信プロトコルの相違を吸収する役割を担う．本論文では，2 章で組込みシステムおよび IoT アプリケーションの開発を説明する．3 章でデバイス用プラグイン開発の効率化を提案し，4 章で提案手法の評価および考察を行う．最後に，5 章で本論文をまとめ，今後の課題を整理する．

2. 組込みシステムおよび IoT アプリケーションの開発

組込みシステムとは，様々な機器に組み込まれ，制御を行うコンピュータシステムである．家電機器や自動車といった多種多様な分野で利用され，私たちの生活と密接に関わっている．対して，IoT は様々なモノがインターネットのように繋がり，情報交換することで相互に制御する仕組みである．様々なハードウェアを扱うため，組込みシステムにおける考え方が重要である．本章では，組込みシステムの特性，モデル駆動開発を説明し，IoT アプリケーションと組込みシステムの類似性および課題，IoT アプリケーション開発における動向を説明する．

2.1 組込みシステムの特性

制御対象である機器が多様であり，機器に組み込まれるシステムも極めて多様となるため，システムにおいて共通する特性は少ない．それがシステム開発を難しくしている一因である．本節では，一般的に組込みシステム開発に共通していると考えられている 4 つの特性を以下に示す[1].

(1) システムが機器に特化していることである．組込みシステムは多様な機器を対象にしており，実装する機能やハードウェア構成はそれぞれ異なる．そのため，各種センサや周辺デバイスといった対象機器に関する環境情報を把握する必要がある．そういった多様な環境情報を用いて最適化しているため，多くのシステムにおいて再利用性の低さが課題である．

(2) リソース制約が厳しいことである．大抵の場合，組込みシステムは必要最低限のリソースしか与えられないことが多く，様々な制約が付随する．

(3) 高い信頼性と品質を保つ必要性がある．組込みシステムでは機械および機器を制御することから，高い信頼性が求められるのが一般的である．大量生産される機器において，システムに問題が発覚した場合，一度販売した機器を回収した後にシステム修正を行う必要がある．回収には多くのコストが掛かり，さらにシステム修正に関してもシステムが製品に搭載されているという性質上，アップデートが困難であるため，莫大なコストを要することになる．そのため，徹底的にシステム検証を行い，高い品質を実現するような開発が必要である．

(4) リアルタイム性が重要になる．組込みシステムが機器を制御する際，大抵の場合に制御する対象の機器により定められた時間条件を満たして動作しなければならない．したがって，開発者は対象機器に定められた時間条件を理解した上で開発する必要がある．

2.2 モデル駆動開発

近年のコンピュータ技術の発展によって，ソフトウェアに対する要求は増し，システムの大規模化および複雑化が進んでいる．そのなかで注目されている手法として，MDD(Model-Driven Development：モデル駆動開発)がある[2]. MDD とは，より抽象度の高い開発を目的としたモデルベースでのソフトウェア開発手法である．モデルとは開発対象に要求される機能をドキュメントやダイアグラムなどを用いて抽象的に表現したものであり，その機能の側面ごとに異なる表記方法のモデルが作成される．MDD において，モデルはプログラミング言語や OS などの実行環境に依存しないモデルである PIM(Platform Independent Model)から，PIM を実行環境に依存するモデルである PSM(Platform Specific Model) に変換していくことで，最終的にはプログラムコードに変換していく．その仕組みを図 1 にまとめる．

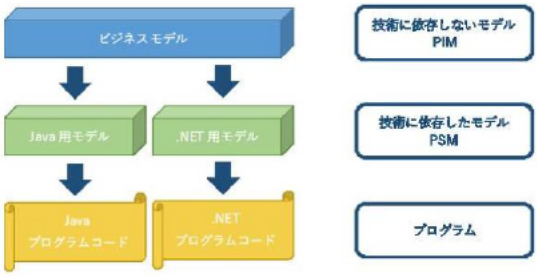


図 1 モデル駆動開発

†和歌山大学大学院システム工学研究科，Graduate School of Systems Engineering, Wakayama University

‡ 和歌山大学システム工学部，Faculty of Systems Engineering, Wakayama University

2.2.1 モデル駆動開発の利点

(1) 保守性の向上である．ビジネス要件とプログラムのプラットフォームを表すモデルをそれぞれ分けることによって，ビジネスと IT 技術を分離することができる．したがって，ビジネスの変化や激しい IT 技術の変化のどちらにも柔軟に対応できるため，保守性が高まる．

(2) 再利用性の向上である．標準化されたモデルの記述方法によって，様々な段階および人との情報共有が進み，再利用性が高まる．

(3) 品質の向上である．開発対象の機能が直感的に理解できるようになり，ノウハウやプロセスが資産化できるため，ソフトウェア生産の効率化や品質を大きく向上することができる．また，自動生成による生産性向上が得られ，モデルは標準化された記述方法を使用するため，そこから生成されたプログラムにはトレーサビリティが備わり一定の品質が保たれる．

2.2.2 モデル

MDD が進展したきっかけは，モデリング言語の普及である．モデリング言語とは図を用いてモデル化を行うものである．代表的なモデリング言語として，UML(Unified Modeling Language)が挙げられる．UML は主にオブジェクト指向分析や設計のための，記法の統一が図られたモデリング言語である．システムの対象機能について図を用いて様々な表現で可視化することができ，様々な利点が存在する．1 つ目は，UML で記述されたモデルは直感的でわかりやすいため，記述方法を知らない顧客なども理解しやすいことである．さらに，全てのプロセスにおいて記法が統一されているため，システム開発において意思疎通を円滑に行うことができる．2 つ目は，ソフトウェア機能を部品化して組み立てることによって変化が容易なシステム作りをできることである．モデルに変更を加えれば自動的にプログラムまで変更が反映できるため，モデルとプログラムの一貫性が保持できる．UML を用いてシステムを実現するためには，システムの動的な構成と静的な構成の両方を明確にすることが重要になる．

2.3 IoT アプリケーション開発の課題

IoT アプリケーションは組み込みシステムの一種であるため，セクション 2.1 で示した組込みシステムの特性に多く当てはまる．さらに IoT アプリケーション開発における課題が 3 つ存在する．1 つ目は，各デバイスメーカーがスマートフォンのネイティブアプリケーション向けに提供する SDK(Software Development Kit)の使い方が統一されていないことである．アプリケーション開発者は，対象のデバイスごとに SDK を開発アプリに組み込む必要がある．2 つ目は，同等の機能を提供するデバイスを利用する場合でも，それぞれデバイスメーカー独自のインタフェースが用意されるため，ソースコードの再利用に限界があることである．3 つ目は，Web ブラウザー上で動作する Web アプリは外部デバイスとの連携に限定されている．

2.4 IoT アプリケーションのためのフレームワーク

セクション 2.3 で示した IoT アプリケーションの課題を解決するために，GotAPI というフレームワークがある[4]．GotAPI とは，標準化団体 OMA(Open Mobile Alliance)にて Generic Open Terminal API Framework Version 1.0 として標準化された仕様書の名前である．Web ブラウザー上で動作する Web アプリや，スマートフォン OS 上で動作するネイティブアプリケーションから，多種多様な外部デバイスへアクセスするためのインタフェースを統一し，Web 技術を用いてスマートフォンと外部デバイスの連携を実現する．GotAPI は HTTP を用いて，外部デバイスとの連携を中継する枠組みを提供する．スマートフォン内部では GotAPI Server という HTTP サーバが起動され，これはひとつのネイティブアプリとしてインストールされるものである．GotAPI Server は REST (Representational State Transfer) ベースの API をアプリに提供する．

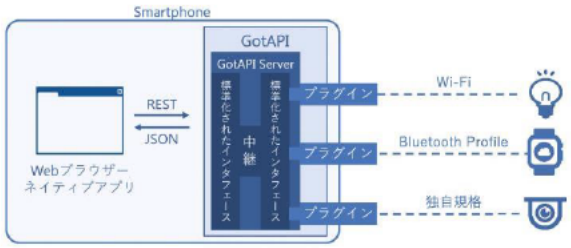


図 2 GotAPI のアーキテクチャ

3. デバイス用プラグイン開発の効率化

本研究は IoT アプリケーション開発を効率化するために，モデル駆動開発手法および GotAPI 仕様の実装例である DeviceConnect の導入を行う．DeviceConnect とは，スマートフォン上に仮想の HTTP サーバを立て，それを介してスマートデバイスを Web API で操作できる GotAPI の実装の一例である．DeviceConnect は，NTT ドコモ社によって開発され，GitHub でオープンソース・ソフトウェアとして MIT ライセンスで提供されている．DeviceConnect は，標準化を先取りして，まだ OMA GotAPI 仕様で規定されていない機能を追加している．例えば，GotAPI 仕様では，非同期にイベントを受信するための枠組みが規定されていないため，DeviceConnect は GotAPI Server に WebSocket サーバとしての役割を追加し，非同期のイベント受信の枠組みも提供している．

3.1 デバイス用プラグイン

GotAPI では，API リクエストとレスポンスを中継する GotAPI Server と，外部デバイスと通信するソフトウェアを分けることで，柔軟にアクセス可能なデバイスを追加できるアーキテクチャになっている．各デバイスへのアクセスは，プラグインというソフトウェアが担う．デバイスとの関わりが深く，機器固有のシステムであるため，多種多様なソフトウェアとなる．プラグインは GotAPI Server と同様に，OS から見れば，1 つのネイティブアプリとしてインストールされるものである．利点は 2 つある．1 つ目は，デバイスごとの通信プロトコルなどの相違を吸収することである．スマートフォンと連携するデバイスの通信方式は多種多様であり，Bluetooth，Wi-Fi といった低レイヤーの通信レイヤーだけでなく，通信手順，データコンテナー，データの値の意味など，あらゆるものが異なる．アプリ開発において，この相違は開発コストに大きな影響を与える．これらの違いをプラグインが吸収し，GotAPI Server に対して，標準化されたインタフェースを用いて結果を返す．アプリは，外部デバイスの通信方式の違いを考慮しなくとも，統一されたインタフェースを使って，外部デバイスと連携

することが可能となる．2 つ目は，デバイスが違っても同じ用途には同じ WebAPI を提供することである．

3.1.1 WebAPI

GotAPI がアプリに提供する API は REST ベースの API であり，それらを総称して WebAPI と呼ぶ． WebAPI のレスポンスは，Web では一般的に使われている JSON を採用している．とりわけ，Web ブラウザー上で動作する JavaScript ベースの Web アプリには親和性が高いといえると考えている．OMA GotAPI 仕様が規定している WebAPI は，アプリの認証やセキュリティー対策などの具体的な API と，実際にデバイスと連携する際に使われるデータのコンテナーを規定している．このデータコンテナーは，アプリと GotAPI サーバの間の通信に加え，GotAPI Server とプラグインとの間の通信も規定している．

3.1.2 ツールを用いたプラグイン作成方法

DeviceConnect では，DeviceConnect Codegen を用いたプラグインの作成方法が示されている．DeviceConnect Codegen は，Swagger の定義ファイルを元にプラグインのテンプレートを出力するツールである．Swagger は REST API を構築するためのオープンソースのフレームワークのことであり，Open API Initiative という団体が REST API のインタフェースを記述するために推進している．大まかな作成手順は以下の通りである．

（1）Swagger 仕様に基づいて JSON 形式で REST API の使用を記述する．
（2）DeviceConnect Codegen を用いて，JSON 形式ファイルから Java などでプログラムのテンプレートを生成する．
（3）2 で生成したテンプレートを編集し，実行内容を記述する．

3.1.3 プラグイン生成手法の提案

モデル変換を用いて，REST API の仕様記述やプラグインコードを導出する流れを図 3 に示す．

（1）デバイスメーカーが提供する SDK(Software Development Kit：ソフトウェア開発キット)に依存しないモデルを作成する．
（2）非依存モデルを依存モデルと REST API の仕様にそれぞれ変換する．
（3）既存ツール(DeviceConnect Codegen)を用いて，REST API の仕様記述からプログラムのテンプレートを生成する．
（4）3 で生成したテンプレートに，依存モデルから導出される実行コードを埋め込む．
（5）プログラム内の導出できない部分は追記する．

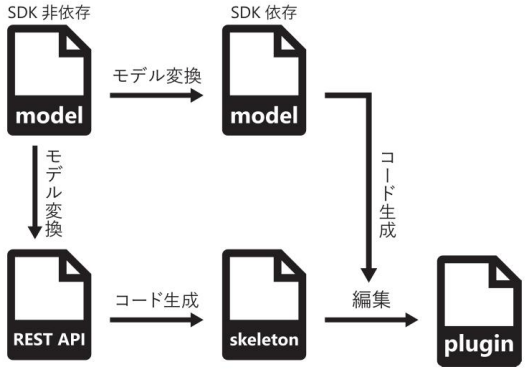


図 3 提案するプラグイン生成手法

3.2 モデル要素とプログラムの関連性

本研究では，モデル要素とプログラムの対応付けが重要になる．分析対象となるプラグインはデバイスにアクセスするための機能だけでなく，Device Connect Manager において単体で動作するための画面設定なども含まれているが，機能のみを対象に検討する．プラグインの機能を定義するプログラムをプロファイルと呼び，主にデバイスにアクセスする役割を持っている．対応付けの方法は，既存プラグインを分析し，モデルに記述し直すリバースエンジニアリングを行った．

3.2.1 Swagger 仕様に基づく REST API 仕様記述の生成手法

モデル要素と REST API 仕様記述の対応付けをまとめる．

（1） x-type は API アクセスデザインパターンを表し，シーケンス図のやり取りから導出する．API アクセスデザインパターンとは，DeviceConnect システムでは，シンプルで手軽に HTTP 通信(REST)と効率的な WebSocket 通信によるイベント処理を行いウェアラブルデバイスや IoT デバイスなどの操作を行う．DeviceConnect システムは以下の 3 つのデザインパターンの基本設計に基づく．

One shot data：シンプルな HTTP 通信(REST)で DeviceConnect Manager にアクセスすることでウェアラブルデバイスや IoT デバイスの情報を取得操作を行う．One shot data の流れを図 4 で表す．

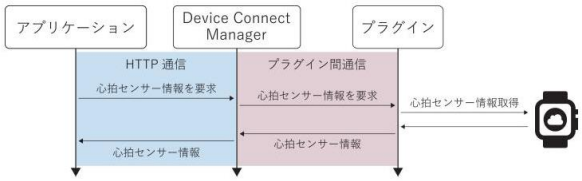


図 4 One shot data

Event driven data：WebSocket を使用してウェアラブルデバイスや IoT デバイスで発生するイベントを連続的に受信する．Event driven data の流れを図 5 で表す．

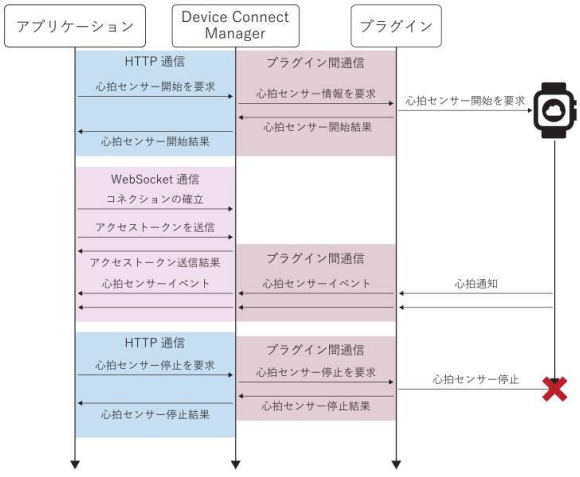


図 5 Event driven data

Streaming data：各プラグインが Web サーバを立ち上げた URI をアプリケーションに中継する．アプリケーションでは受け取った URI を直接参照することで DeviceConnect Manager を経由せずに映像リソースなどのデータを受信する．Streaming data の流れを図 6 で表す．

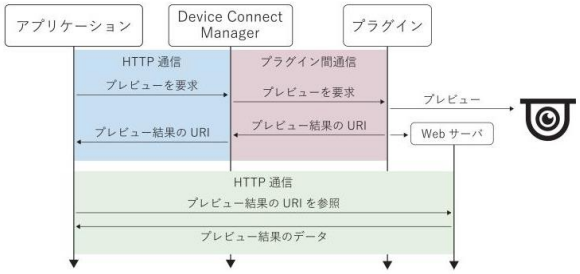


図 6 Streaming data

(2) parameters はリクエストパラメータを表し，クラス図から導出する．
name：変数名を表す．モデルでは属性である．
in：変数の場所を表す．Device Connect を分析した結果，query のみを用いているため，本研究では query のみを対象とする．
required：対象変数の記述が必須であるか表す．シーケンス図にて，プラグイン自身に設定した値は操作に必要な値であるため，必須である．
type および format：変数のタイプおよび型を表す．クラス図では変数の型である．

(3) 個々のデバイスから取得する変数を表す．個々のデバイスによって多種多様なレスポンスが存在するため，具体例を用いる．
1. システムレスポンス
result：処理結果を表す．
product：システム名を表す．
version：システムバージョンを表す．
2. イベントレスポンス
serviceId：サービス ID を表す．
product：システム名を表す．
attribute：アトリビュート名を表す．

3.2.2 プロファイル生成手法

本研究ではリバースエンジニアリングを行っているため，本項では操作を抽象化していくことが主題となる．実際に SDK に依存するモデルから SDK に依存しないモデルに作成すると，対応できる部分とできない部分が発生した．作成したモデルは図 7 と図 8 に示す．対応できる部分は DeviceConnect が定義している一般的な機能を使用している．また，生成できない部分はデバイスメーカー独自の SDK を使用しており，それぞれ使用方法が異なるため，必要なプラットフォームを定義する必要がある．モデルからプラットファームを生成する方法として，配置図を用いて使用方法の手助けにしたいと考えている．デバイスの接続方法が多様化しており，デバイスメーカーの SDK を用いてデバイスの接続を定義する場合，デバイスの情報を記述する必要があるため，デバイス名や通信経路といったデバイスの情報を配置図で表現する．

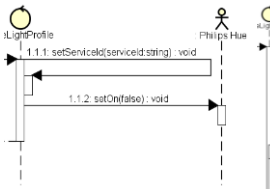


図 7 SDK 非依存モデル

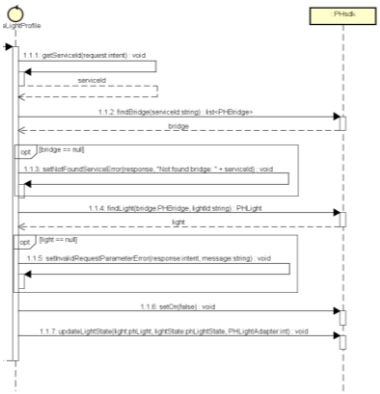


図 8 SDK 依存モデル

4. 提案手法の評価と考察

4.1 コード生成率による評価

本研究の提案手法に基づく REST API の仕様記述およびプラグインに関するコード生成率の評価を行った．対象は Hue および Android Wear を用いた簡易的なシステムである．それぞれのプラグインおよびシステム全体のコード生成に関する評価を表 1，表 2，表 3 に示す．

表 1 Hue のプラグイン生成に関する評価

生成対象	コード生成量（行）	コード全体量（行）	コード生成率（%）
REST API の仕様記述	303	331	92.8
プラグイン	103	158	65.2

表 2 Android Wear のプラグイン生成に関する評価

生成対象	コード生成量（行）	コード全体量（行）	コード生成率（%）
REST API の仕様記述	328	368	89.1
プラグイン	29	72	40.3

表 3 システム全体のプラグイン生成に関する評価

生成対象	コード生成量（行）	コード全体量（行）	コード生成率（%）
REST API の仕様記述	631	699	90.3
プラグイン	132	240	55.0

4.2 生成率と生成対象の関係性

結果として，REST API の仕様記述は比較的コード生成率が高くなり，プラグインについてはデバイスによって差異はあるが，約 5 割程度であった．生成できなかった要素を元に，生成率の要因を考察する．REST API の仕様記述で生成できなかった要素は，変数に追記される情報である「概要」などや，デバイス固有の情報である「変数の最大値および最小値」であった．したがって，動作に関連するコードはほとんど生成でき，REST API の仕様記述に関するコード生成率は比較的高かった．

対して，プラグインでは，主に実装で行われる複数スレ

ッド間の同期やイベント処理をモデルで表現できなかったため，実装に関する記述量が生成率に影響していると考えた．Hue と Android Wear のシーケンス図を用いて，その影響について考察する．図 9 は Hue のシーケンス図，図 10 は Android Wear のシーケンス図を示す．API アクセスデザインパターンの観点において，シーケンス図を比較すると，Hue はデバイスの情報取得や情報更新などすべての場合において，One shot data というシンプルな HTTP 通信で Device Conenct Manager にアクセスすることでウェアラブルデバイスや IoT デバイスの情報を取得操作を行っている．対して，Android Wear はデバイスの情報取得以外の場合において，Event driven data という WebSocket を使用してウェアラブルデバイスや IoT デバイスで発生するイベントを連続的に受信しており，イベント処理が比較的多く含まれている．

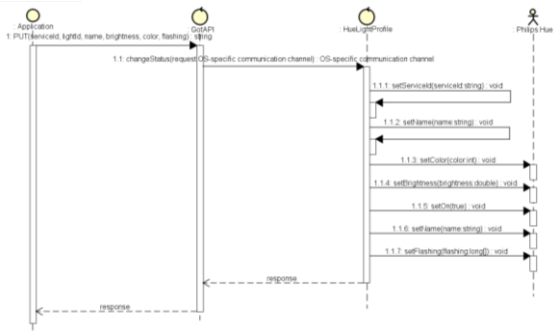


図 9 Hue のシーケンス図

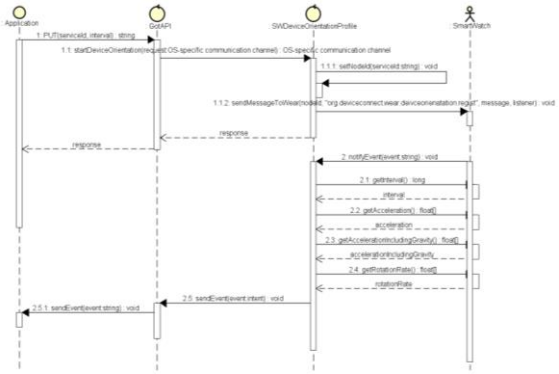


図 10 Android Wear のシーケンス図

4.3 生成に関する分類の妥当性

先行研究として，組込みシステムを対象としたモデル駆動開発に基づくコード生成を行っている[5]．ツールの有効活用を目的としたモデル駆動開発手法を提案しており，具体的なシステムへのツール適用事例の分析によって，「実行環境層」「Bridge 層」「モデル駆動層」の階層構造を持つアーキテクチャを導出している．

(1) 実行環境層はセンサや操作画面など実行環境によって扱う対象が変化する機能の層であり，モデル駆動開発専用のスクリプト言語に対応していない処理が含まれるため，手書きプログラムで実現する．ハードウェアに依存するプログラムであるデバイスメーカーの SDK による記述がこの層に分類される．個々のデバイスによって，SDK の記述は異なるため，手書きプログラムとなる．

(2) Bridge 層は実行環境層とモデル駆動層間のデータやイベントの流れの橋渡しをする機能の層であり，主にゲッターとセッターの集合やイベント発生時に呼び出し先を指定する処理によって構成される．この層は開発ツールでスケルトンコードを生成し，プログラムを加筆することで実現する．DeviceConnects では一般的な機能を定義しているため，非依存モデルを変換することで，ほとんど生成できる．

(3) モデル駆動層はモデル駆動開発専用のスクリプト言語にすべて対応している機能の層であり，主にセンサや操作画面などからのイベント入力によって行われるシステムの内部制御や，制御に必要なパラメータを扱っている．この層は，開発ツールで自動的にプログラムを生成することにより実現する．アプリケーションとハードウェアを接続する際に，DeviceConnect Manager が必要な変換を担う．

5. おわりに

本研究では，IoT アプリケーション開発の効率化を目的とし，MDD に基づく IoT アプリケーション開発におけるプラグイン生成手法を提案した．結果として，プログラム生成率は比較的高いものとなった．REST API の仕様記述は比較的コード生成率が高くなり，対象デバイスによってプラグインの生成率に差異はあったが，約 5 割程度生成できた．プラグインの生成率に差異が出たのは，API アクセスパターンの違いによって実装に関する記述量が異なるためであると考えた．

今後の課題は 2 つある．1 つ目は，モデルから生成できなかった実装に関する記述を詳しく表現する手法を検討する必要がある．2 つ目は，本研究はプラグイン分析手法として，リバースエンジニアリングを行ったため，SDK に依存しないモデルから SDK に依存したモデルを生成できるか検討する必要がある．

参考文献

[1]SESSAME WG 著，組込みソフトウェア開発のためのオブジェクト指向モデリング，翔泳社，2006.

[2]David S. Frankel 著，MDA モデル駆動アーキテクチャ，星雲社，2003.

[3]井上樹 著，ダイアグラム別 UML 徹底活用，翔泳社，2005.

[4]GotAPI ｜ デバイス WebAPI コンソーシアム，https://device-webapi.org/gotapi html

[5]松永純一 著，組込みシステムへの開発ツール適用事例に基づく反復型モデル駆動開発プロセスとモデル記述指針，和歌山大学大学院システム工学研究科，2013.