

RTCOP：組込みソフトウェア開発への適用を考慮した C++ベースのコンテキスト指向プログラミング フレームワーク

谷川 郁太^{1,a)} 久住 憲嗣¹ 小倉 信彦² 菅谷 みどり³ 渡辺 晴美⁴ 福田 晃¹

受付日 2018年11月20日, 採録日 2019年5月9日

概要：本論文では，C++ベースの組込みソフトウェアのためのコンテキスト指向プログラミング（COP）フレームワークを提案する．提案フレームワークは，組込みソフトウェアへの COP 適用を可能とすることを目的としている．Internet of Things（IoT）や Industry 4.0 を代表とする近年の組込みシステムは，コンテキストに動的に適応することが求められている．ここでいうコンテキストとは，システムを取り巻く外部環境や，システムの内部状態，あるいはそれらの変化の順序である．コンテキスト指向プログラミング（COP）は，このようなソフトウェア開発に適した技術である．COP はコンテキストに対応する振舞いを明示的に扱い，実行時にコンテキストの変更に動的に適応するメカニズムを提供する．既存の COP 言語の主なものは，Java や Smalltalk などの拡張である．これらのプログラミング言語は組込みソフトウェア開発には適していない．今でも多くの組込みソフトウェアは C または C++ で開発されており，C++ を COP のために拡張することは重要である．提案フレームワーク RTCOP は，COP 実現のためにメソッドディスパッチ機構を拡張し，レイヤアクティベーションの機構を備えている．さらに，組込みソフトウェアへの適用に十分な性能を実現するために，提案フレームワークには以下の特徴がある．(1) メモリ消費量を抑えるために構成可能な作りである，(2) メソッドディスパッチ・レイヤアクティベーションの速度が実用的なレベルに収まる．本論文では，提案フレームワークの実現方法と性能が妥当な範囲であることについて述べる．

キーワード：コンテキスト指向プログラミング，組込みシステム，動的書き換え，C++

RTCOP: Context-Oriented Programming Framework based on C++ for Applying to Embedded Software Development

IKUTA TANIGAWA^{1,a)} KENJI HISAZUMI¹ NOBUHIKO OGURA² MIDORI SUGAYA³ HARUMI WATANABE⁴
AKIRA FUKUDA¹

Received: November 20, 2018, Accepted: May 9, 2019

Abstract: In this paper, we propose a COP framework for embedded software based on C++. The framework aims to enable us to apply COP to embedded systems. Recent embedded systems are required to dynamically adapt to contexts in the development of IoT or Industry 4.0. The context is the external environment or internal states of a system, or the order of their changes. Context-oriented Programming (COP) is an approach that is suitable for the development of such software systems. COP treats context-dependent behaviors explicitly and provides mechanisms to adapt a behavior of software dynamically to the changes in the context at runtime. Principal existing COP languages are extensions of Java, Smalltalk, etc. These programming languages are not suitable for embedded systems. Many embedded software systems are still developed using C or C++, thus extending C++ with COP features is considered important. Our COP framework named RTCOP has the extension of method dispatching and the mechanism of layer activation to realize COP. Furthermore, to achieve performance acceptable for embedded systems, the framework has the following features: (1) the framework is configurable to reduce memory consumption; (2) the overhead of method dispatching and layer activation is acceptable. In this paper, we describe how to implement our framework and validity of performance.

Keywords: context-oriented programming, embedded system, dynamic rewriting, C++

1. はじめに

近年の組込みシステムは、IoT やインダストリー 4.0 などに代表されるとおり、刻一刻と変化する外部環境に応じて、全体の振舞いを変えるようなシステムを実現する技術が求められている。たとえば、掃除ロボットでは、人が外出したときに清掃を開始し、問題が発生した場合には周囲を撮影してスマートフォンに通知するといった機能が考えられる。

コンテキスト指向プログラミング (Context-Oriented Programming: COP) [1], [2], [3], [4], [5] は、コンテキストに依存したソフトウェアを開発するためのプログラミング技術であり、上記のようなシステムを実現するための技術として注目されている。ここでいうコンテキストとは、システムの振舞いを変える要因であり、プログラムから観測可能なものである [1]。コンテキストとしてあげられるものは、システムを取り巻く外部環境や、システムの内部状態、あるいはそれらの変化の順序があげられる。COP はコンテキストに依存した振舞いをレイヤとしてモジュール化し、それらを実行時のコンテキストの変化に応じて切り替える。これにより、COP はコンテキストに応じて全体の振舞いを変えるようなソフトウェアの開発を容易にする。

これまでの COP の研究で様々な COP 言語が提案されてきた。代表的な COP の調査では [2] が著名であるが、Java や Smalltalk など組込みソフトウェア向けでない言語を拡張したものが中心である。一方、組込みシステム用のプロセッサがサポートする言語は C または、C++ であるため、C++ で COP を使えるようにすることは重要である。

上記のことをふまえ、我々は C++ ベースの組込みソフトウェア向けの COP フレームワーク RTCOP を提案する。提案フレームワークは、C++ の COP 実現のために、メソッドディスパッチ機構を拡張し、レイヤアクティベーションなどの COP に必要な機構を備える。また、組込みソフトウェアへの適用に十分な性能を実現するために、以下の特徴を備える。(1) メモリ消費量を抑えるために構成可能な作りである、(2) メソッドディスパッチ・レイヤアクティベーションの速度が実用的なレベルに収まる。(1) により、COP の各種機能のうち、アプリケーションによっては不要な機能を外すことで、メモリ消費量を節約する。(2) により、性能が求められる組込みソフトウェアへの適

用を可能とする。

本論文では、提案フレームワークの実現方法と性能が妥当な範囲であることについて述べる。妥当な範囲の性能であるかを評価するため、提案フレームワークの実行速度を計測し、我々が想定する組込みシステムの要求と比較することで、実用的なレベルの性能であることを示す。さらに、他の COP 言語と比較することで、提案フレームワークが組込みソフトウェアへの適用に優位であることを示す。

以降、2 章では、関連研究として、既存の COP 言語とそれらの要素技術を紹介する。3 章では提案として、提案フレームワークのシステム要求やプログラムの記述方法を示す。4 章では 3 章の提案フレームワークを実現する方法を示し、それを採用した理由について述べる。5 章で先ほど述べた評価を行い、提案フレームワークが実用的なレベルの性能であることを示す。6 章で本論文での提案をまとめ、今後の課題を述べる。

2. 関連研究

本章では、提案フレームワークの必要性を示すために、2.1 節で COP の概要について述べ、既存の COP 言語をいくつか紹介する。また、3 章、4 章の準備として、2.2 節で COP の要素技術をいくつか紹介する。2.3 節で、COP を採用する理由を明らかにするために、COP と類似の技術であるアスペクト指向プログラミング (Aspect-Oriented Programming: AOP) との違いを示す。

2.1 COP の概要

冒頭で述べたとおり、COP は実行時のコンテキストに応じて振舞いを変化させることが可能である。Robert Hirschfeld らは、COP が対応すべき要素として、振舞いの変種、レイヤ、アクティベーション、コンテキスト、スコーピングをあげている [1]。振舞いの変種は、ベースとなるクラスやメソッドの部分定義であり、レイヤは、これら振舞いの変種をグループ化したモジュールである。既存の COP 言語では、レイヤ上で部分定義されるクラスやメソッドのことをそれぞれパーシャルクラス、パーシャルメソッドと呼んでいる。COP はレイヤを実行時のコンテキストに応じて動的にアクティベーションするフレームワークを持ち、これによって、プログラム中のクラス・メソッドを一度に変更する。コンテキストは、システムの振舞いを変える要因であり、プログラムから観測可能なものである。コンテキストとしてあげられるものは、システムを取り巻く外部環境や、システムの内部状態、あるいはそれらの変化の順序などがある。Robert Hirschfeld らは、上記の論文において、レイヤアクティベーションのスコープを明示的に記述することの必要性を述べているが、COP 言語によっては必ずしもこれが満たされているとは限らない。レイヤアクティベーションの方法の違いについては 2.2 節

¹ 九州大学

Kyushu University, Fukuoka 819-0395, Japan

² 東京都市大学

Tokyo City University, Yokohama, Kanagawa 224-8551, Japan

³ 芝浦工業大学

Shibaura Institute of Technology, Koto, Tokyo 135-8548, Japan

⁴ 東海大学

Tokai University, Minato, Tokyo 108-8619, Japan

^{a)} tanigawa@f.ait.kyushu-u.ac.jp

で述べる。

COP の研究によって、これまでに様々な COP 言語が提案された。これらの言語は、ベースとなるプログラミング言語を拡張することで実現されている [6]。ContextL [5] は、最初に COP の拡張を行った言語である。ContextL は Lisp がベースであり、Common Lisp のオブジェクトシステムを拡張している。それに続き、動的プログラミング言語で COP ライブラリが開発された。これらのライブラリは、ベースとする言語で提供されるメタレベルの機能を用いて実現されている。例として、Smalltalk をベースとする ContextS [7]、JavaScript をベースとする ContextJS [8]、Python をベースとする PyContext [9] などがある。これら動的プログラミング言語以外に、Java を拡張したものとして ContextJ [4]、ContextJ* [1]、JCop [3]、EventCJ [10] などがある。また、C# のアスペクト指向プログラミング言語として LOOM.NET [11] があり、このサンプルプログラムとして、LOOM.NET を用い COP を実現したものがある。ネイティブ環境で動作させることが可能な COP 言語としては、Objective-C をベースとした Subjective-C [12] があげられる。この言語は macOS と iOS を対象としている。

これらのプログラミング言語は、ベースとなるプログラミング言語が組み込みソフトウェア向けではない。組み込みシステム用のプロセッサがサポートする言語は C または、C++ が中心であるため、組み込みソフトウェアへの適用のためには、C++ のサポートが重要である。提案フレームワークの貢献は、COP を組み込みソフトウェアへ適用するために、C++ で COP を実現可能とするところである。

2.2 COP の要素技術

2.1 節で紹介した COP 言語ごとに実装方法が異なる要素技術として以下のものがあげられる [6], [13]。

- メソッドディスパッチ
- レイヤアクティベーション

以降、提案フレームワークを実現する上で重要な上記の要素について、既存の COP 言語で採用されている方法を紹介し、それぞれの利点・欠点について記す。さらに、提案フレームワークでどの方法を採用したかについて述べる。

Malte Appeltauer らは COP のためのメソッドディスパッチの実現手段を 2 つの方法に分類している [6]。1 つはメソッドディスパッチのたびにプロキシオブジェクトを介する方法である。プロキシオブジェクトは、メッセージ受信後にアクティブなレイヤを調べ、実行するメソッドをディスパッチする。もう 1 つはアクティブなレイヤが変更されるたびに、仮想関数テーブルを書き換える方法である。前者はメソッドディスパッチのたびにアクティブなレイヤを調べるためのオーバーヘッドがかかる。後者はレイヤアクティベーションのたびに仮想関数テーブルを変更するためのオーバーヘッドがかかる。

紙名はレイヤアクティベーションの方法と影響範囲によって COP 言語の分類を行っている [13]。レイヤアクティベーションの方法は以下のものがある。

with ブロック：アクティベーション範囲をブロックで囲む方法である。この方法はレイヤがアクティブである範囲を明示的に表せるが、制御フローをまたがるアクティベーションを表現するのが難しい。また、ブロックがプログラム中に散在する問題がある [3]。

決定的なアクティベーション：アクティベーションの命令が用意されており、その命令を実行するとレイヤが永遠にアクティブとなる方法である。制御フローをまたがるアクティベーションが容易だが、意図しないレイヤアクティベーションの衝突が起こる恐れがある [13]。

イベント駆動：青谷らが提案した EventCJ [10] で行っている方法である。この方法ではイベント宣言とレイヤ遷移規則を用いてレイヤアクティベーションを宣言的に指定する。イベント宣言で宣言したイベントが発生した際に、レイヤ遷移規則に応じてレイヤやコンテキストを切り替える。この方法の特徴としては、レイヤアクティベーションの指定が宣言的に行われることでアクティベーションのためのコードが散在しないことや、レイヤ遷移規則を用いることでモデル検査を行いやすく、仕様との一致を検査できることがあげられる。

また、レイヤアクティベーションの影響範囲としては、スレッドごと、あるいはインスタンスごとのアクティベーションやアプリケーション全体の振舞いの変更される大域的なアクティベーションがある。

2.3 COP と AOP の違い

コンテキスト指向プログラミング (COP) は、実行時に書き換えを行うアスペクト指向プログラミング (Aspect Oriented Programming : AOP) であるダイナミックアスペクト指向プログラミング言語 (Dynamic Aspect Oriented Programming : DAOP) の発展形であることから、COP と AOP の違いについて以下に述べる。

AOP はオブジェクトに分割が難しい横断的関心事を扱うプログラミング言語であり、DAOP は実行時に動的にプログラム書き換えを行う AOP である。AOP はポイントカットと呼ぶ箇所に横断的関心事のモジュールを差し込むことで関心事の分離を可能にしている [14]。

AOP は組み込みソフトウェアにおける派生開発など、機能追加の面で導入される事例がいくつか存在する。その一例として、横断的関心事となる機能をアスペクトとして実装し、その機能に変更を加える際に、該当するアスペクトを拡張するような手法が提案されている [15]。

COP と DAOP の違いは、COP はコンテキストに依存する横断的関心事をレイヤとしてモジュール化し、それを動的にアクティベーション・ディアクティベーションする

ための機構を備えているところである。以上のことから、AOP は組込みソフトウェアの機能追加に関する問題を扱うのに対し、COP はソフトウェア実行時のコンテキストによって振舞いを変えるような、近年の組込みソフトウェアの開発に適しているといえる。

3. 提案

本章では提案として、提案フレームワークのシステム要求とプログラムの記述方法を示す。

3.1 システム要求

冒頭で述べたとおり、提案フレームワークは組込みソフトウェア開発で使われやすい C++ を COP 拡張することを目的としている。COP に必要な要素として 2 章で述べたもののうち、言語機構として必要な要素は振舞いの変種、レイヤ、アクティベーションである。提案フレームワークは、これらの要素をプログラム中で記述可能とするためのコンパイラと、これらの機能を実現するためのライブラリを必要とする。また、提案フレームワークでは、これらの要素を組込みソフトウェアに適用できるだけの性能を持つために、以下の特徴を備える。

- (1) メモリ消費量を抑えるために構成可能な作りである
- (2) メソッドディスパッチ・レイヤアクティベーションの速度が実用的なレベルに収まる

これらについて、3.1.1 項、3.1.2 項で述べる。また、2.2 節で述べたとおり、レイヤアクティベーションの方法は 3 種類あるため、どの方法を採用するか決める必要がある。これに関しては 3.1.3 項で述べる。

3.1.1 構成可能性

提案フレームワークは特徴 (1) として、表 1 に示す機能を有効・無効にする仕組みを備える。表 1 の機能のうち、「レイヤ固有メンバ変数」は RTCOP 固有の機能である。表 1 の機能はコンパイル時に機能を無効にすることにより、その機能に対応するプログラムコードや定数、変数の定義をコンパイラから生成するソースコードや COP 機能を提供するライブラリから削除する。これにより、メモリ消費量を削減する。

proceed は多くの既存の COP 言語が備えている機能であり、アクティブなレイヤが複数あるときに、あるレイヤのパーシャルメソッドから別のレイヤのパーシャルメソッドやベースメソッドを実行する。たとえば、JCOP におい

て α , β というレイヤを順番にアクティベートした場合、メソッドディスパッチ時には、初めにレイヤ β のパーシャルメソッドが実行され、その中で proceed を用いると、レイヤ α のパーシャルメソッドが実行される。同様にレイヤ α で proceed を行うとベースメソッドを実行する。proceed は複数のレイヤを扱う際に重要な機能だが、この仕組みを実現するためには、レイヤ間の関係を意識したアクティベーション・メソッドディスパッチが必要となるため、実行時間やメモリ使用量の増加につながる。

レイヤ固有のメンバ変数は、カプセル化によってレイヤの再利用性向上が期待できる。一方で、インスタンス化時に、この変数を扱うためのメモリ領域の確保や、プログラム中で使用可能にするための準備が必要になるため、メモリ使用量や実行時間の増加が問題である。

レイヤアクティベーションイベントハンドラは、レイヤアクティベーション・ディアクティベーション時に、ユーザ定義の振舞いを実行させるために必要であるが、すべてのレイヤで必要となるとは限らない。必要ないレイヤでは、イベントハンドラの機能を無効にすることで、メモリ使用量を節約する。

3.1.2 性能

(2) は性能が求められる組込みソフトウェアへの適用を可能とするために重要である。COP の性能決定の要因として、2.2 節で述べた COP のためのメソッドディスパッチの実現方法があげられる。2.2 節で述べたとおり、プロキシオブジェクトを介する方法はメソッドディスパッチ時にオーバーヘッドがかかり、仮想関数テーブルの書き換えではレイヤアクティベーション時にオーバーヘッドがかかる。

我々は、メソッドディスパッチにかかるオーバーヘッドを削減するために、後者の方法を採用した。組込みシステムの振舞い変更では、使用するハードウェアの切り替えがボトルネックとなり得るため、レイヤアクティベーションの時間よりもメソッドディスパッチの時間を削減するほうが効果的である。

3.1.3 レイヤアクティベーションの方法

2.2 節で述べたとおり、レイヤアクティベーションの方法は 3 種類ある。我々は、イベント駆動で影響範囲が大域的なアクティベーションを目指している。この方式は、レイヤアクティベーションやコンテキスト推定が散在しないため、近年組込みシステム開発で流行している分散システムへの適用に向いている [16]。

提案フレームワークでは、イベント駆動方式の前の基礎として、決定的なアクティベーションを採用している。この方式は、宣言的ではなくプログラム中にアクティベーション命令を直接記述するところがイベント駆動と異なる。イベント駆動よりも実装しやすく、任意のタイミングでアクティベーションが行えるために性能評価を行いやすいため、今回の提案で採用した。将来的には、イベント駆

表 1 無効化可能な COP の機能の一覧

Table 1 Removable COP functions.

機能	概要	他COP言語の実装状況
Proceed	パーシャルメソッド中で実行可能な特別なメソッド ベースメソッドや他のアクティブなパーシャルメソッドを実行	ほぼ全てのCOPで実装
レイヤ固有メンバ変数	あるパーシャルクラス・メソッドでのみ利用可能なメンバ変数	RTCOP固有
レイヤアクティベーションイベントハンドラ	レイヤアクティベーション・ディアクティベーション時の 初期化・終了処理を行うためのイベントハンドラ	EventCJIに類似機能有り

動によるアクティベーション記述を追加することで、イベント駆動方式にすることを狙っている。

3.2 プログラムの記述例

本節では、3.1 節で示した COP の要素や、提案フレームワークが備える固有の機能を用いた際のプログラムを示し、提案フレームワークの持つ機能を明確にする。

図 1 はベースクラスとレイヤの定義、レイヤアクティベーションによる振舞い変更のプログラム記述例である。これらのうち、レイヤ定義はレイヤ記述というプログラムで専用の記法を用いることで行う。これは、既存資産の活用や学習コストの軽減を目的としている。レイヤの定義以外は C++ プログラムとして記述する。また、レイヤではアクティベーション時に実行されるイベントハンドラを定義することもできる。これは、アクティベーション時に実行したいメソッドに RTCOP 専用のアノテーションを付けることで行う。

図の例は Hello というベースクラスを C++ で定義している。メソッドディスパッチの実現方法の関係から、ベースメソッドの Print は仮想関数である。レイヤ記述では、Japanese というレイヤを定義しており、その中で Hello クラスのパーシャルクラスを定義している。パーシャルクラスでは、レイヤ固有のメンバ変数や Print() メソッドのパーシャルメソッドを定義している。ここで表 1 の機能について使わないものについては、外すことが可能となっている。

main() メソッドでは、4 章で述べる実装方法の関係上、Hello クラスのインスタンス化時に copnew という特別な機能を使っている。その後の行でレイヤアクティベーションの前後のメソッドディスパッチの結果や、アクティベーション時のイベントハンドラの実行について示している。ここで、copnew でなく C++ の new を用いた場合や、スタックに確保したオブジェクトについては、振舞い変更を

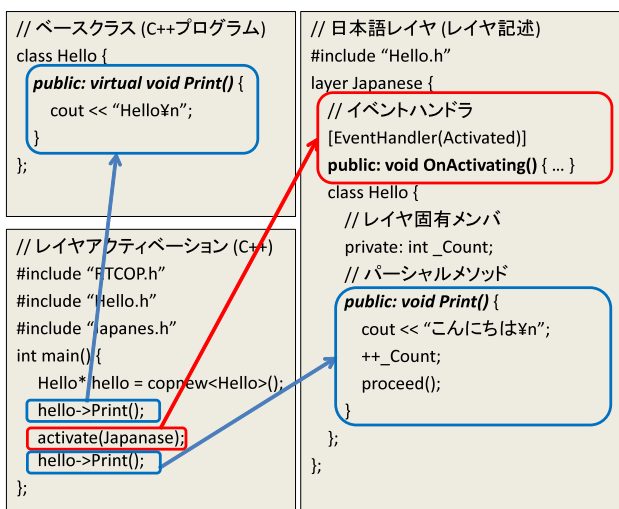


図 1 RTCOP のプログラム記述例
Fig. 1 Programs of RTCOP.

反映しない。また、ヒープ領域を使いたくない場合の代替案としては、new 演算子のオーバーロードと組み合わせること、静的領域に割り当てる方法などが考えられる。

4. 実装

本章では、3 章で示した提案フレームワークの実装方法を示す。4.1 節では、提案フレームワークの構造を示し、以降の節でその中の各要素の実装について述べる。

4.1 提案フレームワークの構造

図 2 に提案フレームワークの使用プロセスを示す。提案フレームワークは 3.2 節で示したレイヤ記述を C++ プログラムに変換するためのレイヤ記述コンパイラと、COP 実現のためのメソッドディスパッチ拡張や、レイヤアクティベーション機構の提供を行う COP ライブラリによって構成される。前者はレイヤ記述によってレイヤの定義を簡単にすることを目的とし、後者は COP の機能を提供することを目的とする。

これらのプログラムを一般的な C++ コンパイラによってコンパイル、リンクすることで、実行可能とする。組み込みソフトウェアの開発では、対象とするマイコンボードごとにコンパイラが異なることが一般的であるため、COP プログラムから直接アセンブリを生成するコンパイラを作るには手間がかかる。レイヤ記述コンパイラがレイヤ記述のみを対象とする理由は、既存資産を再利用しやすいためである。すべてのプログラムをレイヤ記述に書きなおし、コンパイルしなおすことは手間がかかるため、ベースクラスのプログラムは直接 C++ コンパイラでコンパイルする。

3.1 節の特徴 (1) のため、提案フレームワークはレイヤ記述で使われている機能に応じて、レイヤ記述コンパイラによる変換後のソースコードを変化させる。また、どの COP ライブラリを読み込むか自動で選択するツールを持つ。

以降、4.2 節で COP ライブラリの実装について、4.3 節でレイヤ記述コンパイラの実装について述べる。

4.2 COP ライブラリ

COP ライブラリの主な役割として、メソッドディスパッチの拡張とレイヤアクティベーション機構の提供がある。また、オプションとしてレイヤ固有メンバ変数やレイヤア

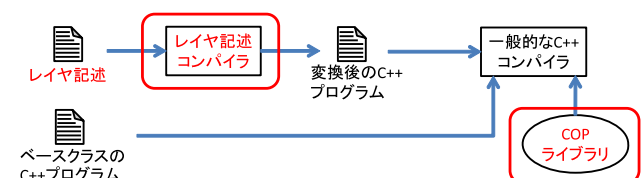


図 2 RTCOP の使用プロセス
Fig. 2 Process of RTCOP programming.

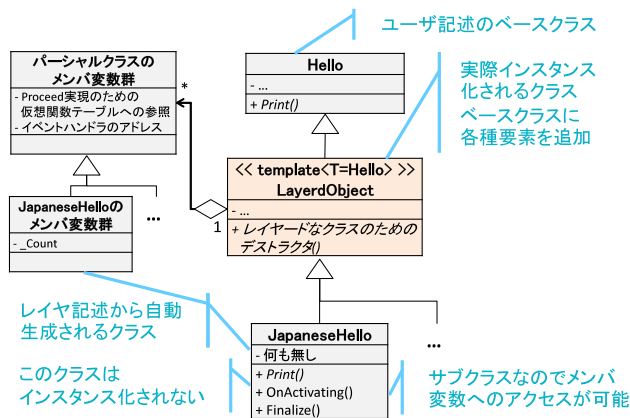


図 3 レイヤードなクラスの構造

Fig. 3 A structure of a layerd class.

クティベーションイベントハンドラの機能を扱う。これらの機能を実現するために、レイヤによって振舞いを変える可能性のあるクラスをインスタンス化する際の工夫がある。本論文では、このようなクラスのことをレイヤードなクラスと呼ぶこととする。

これらの実現方法について述べるために、4.2.1 項でレイヤードなクラスのインスタンス化の仕組み、4.2.2 項でメンバ変数の扱い、4.2.3 項でメソッドディスパッチの仕組み、4.2.4 項でレイヤアクティベーションの仕組みについて述べる。また、提案フレームワークは C++ を用いたネイティブな実装であるため、ターゲットとなるコンパイラや OS などに依存する実装部分が存在する。これについて 4.2.5 項で述べる。

4.2.1 レイヤードなクラスのインスタンス化の仕組み

提案フレームワークにおいて、レイヤードなクラスのインスタンス化では、ベースクラスの代わりに、COP 実現のための変数やメソッドを追加したテンプレートクラスのインスタンス化を行っている。このように、実際にインスタンス化するクラスを違うものに置き換える方法は、LOOM.NET のようなベース言語のメタレベル機能を用いる言語で良く使われている。代わりのクラスをどのように実現するかは選択肢はベース言語により異なる。提案フレームワークでは、C++ においてテンプレートがメタプログラミングのために用いられやすいため採用することとした。

レイヤードなクラスの構造を図 3 のクラス図に示す。レイヤードなクラスは図中の LayerObject テンプレートクラスで実現している。このクラスはベースクラスの Hello を型パラメータとして受け取り、さらに継承することで、COP 実現のために必要なメンバ変数やメソッドを追加している。レイヤードなクラスの実現をベースクラスの継承によって行うことで、ベースクラスの元々の継承関係を壊すことなく機能追加を行っている。図 4 の (a) のようにベースクラスから LayerObject を継承すると多重継承と

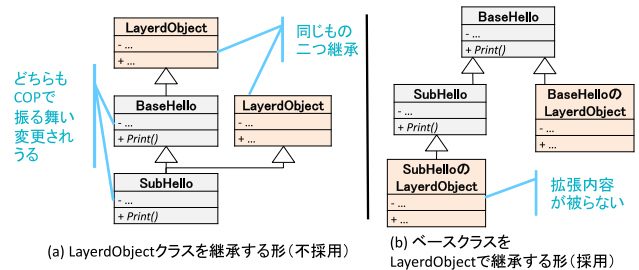


図 4 LayerObject の継承ツリーの案

Fig. 4 Examples of inheritance tree for LayerObject.

なり、問題である。JapaneseHello は、パーシャルクラスの振舞いを実装するために、レイヤ記述から自動生成されるクラスである。パーシャルメソッドを実装することのみが目的であり、このクラスのインスタンスが作られることはない。

図 4 の (b) において、SubHello の LayerObject から BaseHello の LayerObject をスーパークラスのメソッド呼び出しにより実行した場合、LayerObject メンバのアドレスが SubHello のサイズ分だけずれる。このような問題を解決するため、LayerObject のメンバへのアクセスは、継承元のクラスサイズからアドレスを求めることで行う。

4.2.2 メンバ変数の扱い

レイヤ記述において定義されたレイヤ固有のメンバ変数は、図 3 のようにパーシャルクラスのメンバ変数を管理するためのクラスに持たせる。この理由は、JapaneseHello に直接変数を追加すると、LayerObject クラスのインスタンスの有効範囲外のアドレスにアクセスする必要があるためである。このクラスはレイヤードなクラスで持つべき共通の要素を扱うためのクラスを継承している。

パーシャルクラスのメンバ変数群クラスの内容は、表 1 の各種機能の有効・無効で変わる。たとえば、proceed が必要ない場合、そのために必要なメンバを含めないようにする。また、レイヤ固有のメンバ変数が必要ない場合は、パーシャルクラス固有のメンバ変数群クラスを定義しない。

4.2.3 メソッドディスパッチの仕組み

3.1.2 項で述べたとおり、メソッドディスパッチは仮想関数テーブルを書き換えることにより実現する。C++ など多くの OOP 言語では、ポリモーフィズムを実現するための手段として、インスタンスごとに仮想関数テーブルを持っており、メソッドディスパッチの際にテーブルから対応するメソッドのアドレスを取得することで、切り替えを行っている。提案フレームワークでは、copnew によるインスタンス化の際に、仮想関数テーブルのポインタの参照先を提案フレームワークが管理する仮想関数テーブルのアドレスに変更する。提案フレームワークの持つ仮想関数テーブルは、C++ オリジナルの仮想関数テーブルをコピーしたものである。この専用の仮想関数テーブルをアクティベーションの際に書き換えることで、実行するメソッドの

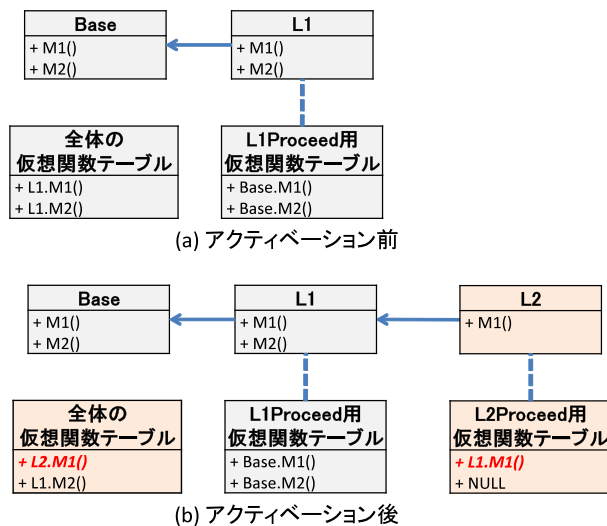


図 5 アクティベーション前後の仮想関数テーブル

Fig. 5 A virtual function tables before and after activation.

切り替えを実現している。copnew の手順としては、最初に C++ の new でインスタンス化を行い、その後に生成したインスタンスの仮想関数テーブルの先頭を指すポインタを変更する。

オリジナルをコピーした仮想関数テーブルを用いる理由は、直接オリジナルの仮想関数テーブルを書き換えることによって中身を破壊しないためである。COP システムでは、実行中のレイヤアクティベーション・ディアクティベーションによって、ディスパッチするメソッドが動的に変化するため、オリジナルの仮想関数テーブルは元の状態のまま保持する必要がある。

提案フレームワークでは、上記の仮想関数テーブルのほかに、レイヤごとに proceed を実現するための仮想関数テーブルが別にある。これらの仮想関数テーブルも C++ オリジナルの仮想関数テーブルをコピーしたものである。この仮想関数テーブルは proceed 実行時に参照され、テーブル中のアドレスで示されるメソッドを実行する。

4.2.4 レイヤアクティベーションの仕組み

3.1.3 項で述べたとおり、レイヤアクティベーションは決定的なアクティベーションを採用している。そのため、アクティベーション命令が発行された時点で、仮想関数テーブルの書き換えが全体に及ぼされる。本項では、レイヤアクティベーション時の仮想関数テーブル書き換えの実装方法を示す。この方法は C++ 以外でも実現可能である。

COP 実現のための全体の仮想関数テーブルと、proceed のためのレイヤごとの仮想関数テーブルの書き換えのルールを明らかにするために、図 5 にアクティベーション前後の上記仮想関数テーブルを示す。図の例では、L1 というレイヤがすでにアクティブであり、L2 というレイヤを新たにアクティベートした際の仮想関数テーブルの変化を表している。各仮想関数テーブルは、メソッド M1、M2 のア

ドレスを持っており、アクティベート順序とパーシャルメソッドの有無に基づいて変化する。

これらのレイヤはアクティベート順に基づきリスト構造で連結される。L2 のアクティベーション時に、前のレイヤをたどっていき、proceed 用の仮想関数テーブルや、全体の仮想関数テーブルの書き換えを行う。全体の仮想関数テーブルは、一番初めに実行されるメソッドのアドレスを保持する。パーシャルメソッド中で proceed 命令が実行された場合は、レイヤの proceed 用仮想関数テーブルを見ながら、次のパーシャルメソッドへ進んでいく。proceed を使わない場合、proceed 用の仮想関数テーブルは用意せず、全体の仮想関数テーブルの書き換えのみを行う。

4.2.5 ターゲットに依存する実装

提案フレームワークには、ターゲットとなるコンパイラや OS に依存する実装がある。具体的には、ユーザが定義したベースクラスやパーシャルクラスのメソッドのアドレスを取得するところがあげられる。

上記については、C++ の標準の機能だけで静的なクラス情報から取得することができないため、アセンブリ言語で実装する必要がある。その際、仮想関数や仮想関数テーブルのシンボル名がターゲットとなるコンパイラや OS によって異なるため、上記はターゲットに依存することとなる。

4.3 レイヤ記述コンパイラ

本節では、レイヤ記述コンパイラがレイヤ記述を基に生成するプログラムの内容を示し、レイヤ記述コンパイラの役割について明らかにする。レイヤ記述コンパイラはレイヤ記述から以下の C++ プログラムを自動生成する。これらのプログラムは、レイヤ記述によって内容が変わる。

- (a) パーシャルクラスの定義 (例：図 3 JapaneseHello)
- (b) レイヤクラスの定義 (例：図 5 レイヤリスト)
- (c) レイヤ数、ベースクラス数、メソッド数などのアプリケーション情報の設定と RTCOP の初期化処理

(a) に関して、図 3 のレイヤ記述から自動生成された C++ プログラムを図 6 に示す。レイヤ固有メンバは 4.2.1 項で述べたとおり、パーシャルクラスメンバ用のクラスに追加される。proceed は 4.2.2 項や 4.2.3 項で述べたとおり、レイヤごとの仮想関数テーブルのアドレスから次のパーシャルメソッドを実行する。

(b) では Layer クラスを継承し、レイヤごとの初期化処理やイベントハンドラを追加する。Layer クラスは図 5 で示したとおり、アクティベートした際にリストにつながれるようになっている。また、このクラスは proceed 時に実行するメソッドの情報を示す仮想関数テーブルを持つ。その他、現在のアクティブ状態やメソッドディスパッチ時の実行順序といった情報を持っている。

(c) の情報はレイヤアクティベーションでどこまで書き

```

//レイヤクラスの定義
class Japanese : public Layer {
    //レイヤの初期化・終了処理など ...
    // イベントハンドラ
    public: virtual void OnActivating() { ... }
};
// パーシャルクラスメンバの定義
class JapaneseHelloMembers : public PartialClassMembers {
    public: int _Count;
};
// パーシャルクラスの定義
class JapaneseHello : public LayerObject<Hello> {
    // パーシャルメソッド
    public: virtual void Print() {
        cout << "こんにちは\n";
        ++_PartialClassMembers[1]->_Count;
        _PartialClassMembers[1]->VTable->Print(); // Proceed();
    }
};

```

図 6 パーシャルクラス定義プログラム

Fig. 6 A program of partial class.

換えるかということや、レイヤ情報を管理するための配列をどのくらいのサイズにするか、といったことをフレームワーク側で把握するために必要である。

これらのプログラムはレイヤ記述の内容に基づいて生成されるものであり、レイヤ記述コンパイラから自動生成する。

5. 評価

本章では、提案フレームワークの性能が妥当な範囲であることについて述べる。以降、5.1 節で我々が想定する組込みシステムと性能要求について述べる。5.2 節で 5.1 節の性能要求を満たすことを示す。5.3 節で、他の COP 言語と比較することで、提案フレームワークの優位性を示す。5.4 節で、構成可能にしたことによる効果を示すために、COP の各機能を実現するために必要なヒープメモリ使用量を示す。

ベンチマーク環境は表 2 のとおりである。開発対象は COP 言語の 1 つである Subjective-C と比較を行うために MacOS とする。

5.1 想定する組込みシステムと性能要求

我々が想定する組込みシステムは、冒頭で述べた掃除機ロボットのように、稼働を止めずに実行時のコンテキストによって全体の振舞いを変更することが求められるシステムである。組込みソフトウェアにおける振舞い変更では、ハードウェアの切り替えも行われることが想定される。提案フレームワークでは、最低限の目標として、ハードウェアの切り替えよりも早い程度の性能を目指す。ハードウェア切り替えの目安として、FPGA のハードウェアの再構成

表 2 ベンチマーク環境

Table 2 Benchmark environment.

OS	macOS High Sierra (バージョン10.13.6)
CPU	1.86 GHz Intel Core 2 Duo
メモリ	4 GB 1067 MHz DDR3
開発環境	Xcode バージョン 10.1 (10B61)
	clang バージョン 4.2.1

にかかる時間は 10 ミリ秒単位である。これは、レイヤアクティベーションにおいて従来の COP と同程度の性能があれば十分に達成できる。

また、メソッドディスパッチにかかる時間は、組込みソフトウェアにおける周期実行を守るために重要である。我々の想定では、周期はミリ秒単位で設定し、1 回の周期でメソッドディスパッチが 100 回以内であるとする。目標はメソッドディスパッチを 100 回実行したときの時間が 1 ミリ秒を超えないこととする。

5.2 性能要求に対する評価

本節では、レイヤアクティベーションとメソッドディスパッチにかかる実行時間を計測し、提案フレームワークが妥当な範囲内の性能であることを示す。メソッドディスパッチについては、C++ と性能比較を行うことで、どの程度性能に影響を与えているかも明らかにする。計測には、sys/time.h の gettimeofday 関数を用いる。また、コンパイラによる最適化は無効とする。

具体的な評価方法は以下のとおりである。準備として、1 つのベースクラスと、3 つのレイヤとそれらのレイヤに対応するパーシャルクラスを用意する。ベースクラスは、int 型のメンバ変数と、それをインクリメントするベースメソッドを持つ。また、パーシャルクラスは上記メソッドをベースとするパーシャルメソッドを持つ。パーシャルメソッドで実行される処理もベースメソッド同様に、ベースクラスが持つメンバ変数のインクリメントである。

手順としては初めに、0～3 つのレイヤをアクティベートし、レイヤードなクラスをインスタンス化する。その後、メソッドを呼び出してから返ってくるまでの処理を 100 万回行う。この 100 万回のループを 10 回繰り返して、1 回ごとにかかる時間を計測する。計測結果を 100 万で割ることによって、1 回のメソッドディスパッチにかかる時間を割り出す。最終的な結果は 10 回の繰り返しの平均とする。また、結果の偏りを明らかにするために、標準偏差も示す。また、ベース言語からの増加率を調べるために、上記の性能評価をベース言語である C++ の仮想関数のメソッドディスパッチに対しても行う。レイヤの代わりに子のクラスを 1 つ用意し、パーシャルメソッドの代わりに、仮想関数の呼び出し・実行にかかる時間を計測する。

レイヤアクティベーションの性能評価では、1 つのレイ

表 3 メソッドディスパッチの計測結果

Table 3 Performance of method dispatching.

	レイヤ数	実行時間 [μ s]	標準偏差 [μ s]	ベース言語からの増加率 [%]
RTCOP(提案)	0	0.0067991	0.0010495	6.198
	1	0.0067864	0.0011961	5.999
	2	0.0067725	0.0010384	5.782
Subjective-C	0	0.0107200	0.0014420	7.351
	1	0.0105930	0.0017918	6.080
	2	0.0106430	0.0010346	6.580

表 4 レイヤアクティベーションの計測結果

Table 4 Performance of layer activation.

	メソッド数	実行時間 [μ s]	標準偏差 [μ s]
RTCOP(提案)	0	0.2268007	0.0029326
	1	0.2607954	0.0044447
	2	0.3384917	0.0044955
	3	0.4086132	0.0034690
Subjective-C	0	13.0228241	0.0208673
	1	346.6672263	0.3720033
	2	494.5269775	0.6213537
	3	645.5425111	2.2963909

ヤに対して、アクティベーションとディアクティベーションを1回ずつ実行し、完了するまでの時間を計測する。この計測でも上記と同様に、100万回ループを10回繰り返して、平均値と標準偏差を示す。また、計測に使うクラスが持つメソッドの個数は0から3つまでとし、それぞれの個数におけるアクティベーション性能の違いも明らかにする。

上記の計測結果を表3、表4に示す。表3から、メソッドディスパッチにかかる時間はC++の仮想関数のディスパッチから6%ほど増加している。どちらも仮想関数テーブルによってメソッドディスパッチを行っているが、元々の仮想関数のアドレスから離れるため、キャッシュミスなどが起こりやすくなり、その分実行時間が増加したことが考えられる。また、レイヤ数はメソッドディスパッチの実行時間に影響を与えないことが分かった。表4から、レイヤアクティベーションにかかる時間は、メソッドの個数が1つ増えるたびに大体0.04から0.07 μ sほど増加する。

提案フレームワークが妥当な範囲内の性能であるかについては、ベンチマーク環境と組み込みソフトウェアの実行環境との差を考慮する必要がある。表2より、ベンチマーク環境はCPUのクロック周波数が1.86GHzだが、組み込みシステムでは数MHzということもあり得る。仮に1.86MHzのクロック周波数であると考えた場合、同じ処理を行うのに1000倍の時間がかかる。表3、表4の結果に1,000倍した場合、メソッドディスパッチはさらに100倍しても約0.68msである。レイヤアクティベーションはパーシャルメソッドの個数が増えるたびに増加する時間を0.07msとすると、1つのレイヤにパーシャルメソッドが100個あったとしても7ms程度である。5.1節で示した性能を満たしていることから、提案フレームワークが妥当な範囲内の性

能であるといえる。

5.3 他のCOP言語と比べた際の優位性

本節では、他のCOP言語と機能、性能を比較することで、提案フレームワークの組み込みソフトウェア開発における優位性を明らかにする。

5.3.1 機能面の比較

機能面について、COP言語に必要な要素は3章で述べたとおり、振舞いの変種、レイヤ、アクティベーションである。提案フレームワークでは、振舞いの変種としてパーシャルクラス・メソッドを宣言・定義でき、それらをまとめるモジュールとしてレイヤを記述できる。また、レイヤをアクティベート・ディアクティベートするための命令や、多くのCOP言語とに用意されているproceed命令を備える。これらのことから、COP言語に必要な最低限の機能を備えているといえる。

他のCOP言語と比べると、提案フレームワークには、2.2節で示したイベント駆動方式のレイヤアクティベーションのような、コンテキストを判断し、レイヤをアクティベーションするための専用のモジュールはない。これについては、今後の課題としている。

表1の機能をすべて無効にした場合にCOP言語であるといえるのかについては、proceed以外の機能は提案フレームワーク固有の機能であったり、ほとんどのCOP言語にはない機能なので、無効にしても他のCOP言語の記述能力には劣らない。proceedについては、無効にすることで複数のレイヤを組み合わせることが困難となり、他のCOP言語に比べ記述能力が劣ることとなるが、RAMの削減効果が見込まれるため、アプリケーション次第では有用である。

その他機能面で気をつけたいこととして、提案フレームワークがベースメソッドとして扱えるのは、仮想関数だけである点があげられる。この制約のために、仮想関数以外の関数に対してレイヤの振舞いを追加できない。これについて、我々は元々のC++が仮想関数の変更しか認めていないため妥当であるという立場をとっている。

5.3.2 性能比較

他のCOP言語との性能比較として、5.2節で示した方法により、Subjective-C[12]の性能評価を行う。Subjective-CはObjective-CをベースとしたCOP言語である。Objective-Cはネイティブコンパイラを前提とし、メモリ管理をガベージコレクションではなく参照カウンタ方式のautoreleaseを標準ライブラリで使用する。これらのことから、動的性質が強い言語ではあるのでC++よりはる程度は性能面で劣る傾向があるが、他のCOP言語よりも比較的性能が良いことが期待できるため、提案フレームワークとの比較対象とした。Subjective-Cの計測には、提案フレームワーク同様に、sys/time.hのgettimeofday関数を用いる。

表 5 機能の有効・無効化による性能比較
Table 5 Performance comparison for each functions.

	ヒープメモリ使用量[Byte]: ポインタ数		
	proceed	イベントハンドラ	全体の使用量
RTCOPが管理する 仮想関数テーブル			4×ベースクラス数+ 4×ベースメソッド数 : クラス数+メソッド数
Proceed実現のための 仮想関数テーブル	4×ベースクラス数+ 4×ベースメソッド数 : クラス数+メソッド数		4×ベースクラス数+ 4×ベースメソッド数 : クラス数+メソッド数
レイヤごとに 必要な情報	4:1	16+(4×イベントハンドラ数) : 4+イベントハンドラ数	56+(4×イベントハンドラ数) : 11+イベントハンドラ数
レイヤードなクラスの インスタンスごとに 必要な情報	4×レイヤ数: レイヤ数		12+(12×レイヤ数) : 2+(3×レイヤ数)
その他アプリケーション 全体で必要となる情報			52+(4×ベースクラス数): 11
合計	4×(クラス数+メソッド 数)+(インスタンス数+1) ×(4×レイヤ数) : クラス数+メソッド数 +(インスタンス数+1)× レイヤ数	(16+(4×イベントハンドラ 数))×レイヤ数 : (4+イベントハンドラ数)× レイヤ数	52+(56+(4×イベントハンドラ 数))×レイヤ数+(12+(12×レイヤ 数))×インスタンス数+12×クラス 数+8×ベースメソッド数 : 11+(11+イベントハンドラ 数)+(2+3×レイヤ数)×インスタ ンス数+3×クラス数+2×クラス数

上記の結果を表 3, 表 4 に示す. メソッドディスパッチについては, 提案フレームワークが $0.004\mu\text{s}$ 早く, ベース言語からの増加率も少ない. レイヤアクティベーションについては, RTCOP の方が実行時間が短く, メソッドが増えたときの増加時間も少ないため, 性能が上である.

これらの結果について, Subjective-C の実装では, Objective-C のリフレクション機能を用い, レイヤアクティベーション時にメソッドのディスパッチ先を決定しているため, メソッドディスパッチにかかる時間は短めである. 一方, レイヤアクティベーションでは, すべてのメソッドのディスパッチ先の決定を Objective-C のリフレクション機能を用いて行っているため, 時間がかかることが考えられる. 提案フレームワークでは, 多くの COP 言語のように, ベース言語に備えられているリフレクション機能は使わずに, 仮想関数テーブルの書き換えを行っているために, Subjective-C よりも良い性能が出たと考えられる.

上記の結果から, 提案フレームワークは, レイヤをアクティベーションするための専用のモジュールがないこと以外は, 他の COP 言語と同程度の記述能力を備え, 性能評価はネイティブ環境で動く COP 言語の Subjective-C よりも良い結果である. このことから, 他の COP 言語よりも組込みソフトウェアへの適用に向いている.

5.4 各機能におけるヒープメモリ使用量

本節では, 構成可能にしたことによる効果について評価するために, 提案フレームワークの各機能を実現するために必要なヒープメモリ使用量を示す. これから示すヒープメモリの使用量は, Visual Studio のデバッガによって正しいことを確認した. 計測は, RTCOP のための初期化処

理が完了し, すべてのレイヤのアクティベーションとレイヤードなクラスのインスタンス化が完了した時点での, 各インスタンスのメモリ使用量を調べることで行った.

表 5 に各機能のヒープメモリ使用量をまとめる. メモリ使用量は x86 のものである. ポインタ変数のサイズは OS によって異なるため, ポインタの個数も併記する. 結果はベースクラス・メソッド数, レイヤードなクラスのインスタンス数, レイヤ数, イベントハンドラ数により変化する. これらの数が多いほど, 機能削減の効果が大きくなる.

次に, 具体的な削減効果を示すために屋内外動く掃除機ロボットの事例を紹介する. このロボットは屋内用と屋外用のレイヤを備えている. これらのレイヤは排他関係にあり, 片方のレイヤをアクティベートしたとき, もう片方のレイヤは必ずディアクティベートする. また, それぞれのレイヤはアクティブ化時, ディアクティブ化時に実行されるイベントハンドラを備える. このイベントハンドラにより, 使用するデバイスを屋内外で切り替えることが可能となる. クラスはロボットの移動方向を決定するためのクラスと, 掃除の処理を行うためのクラスがあり, それぞれ 2 つのベースメソッドを持つこととする. また, 屋内外のレイヤには, これらのベースメソッドの振舞いを変更するためのパーシャルメソッドがある. パーシャルメソッドでは, レイヤごとに共通の処理を proceed の呼び出しにより実行している.

上記の例において, 掃除機ロボット起動時にすべてのデバイスを初期化完了するように変更することで, イベントハンドラが必要なくなる場合について考える. レイヤごとのイベントハンドラの数, アクティベート時のものとディアクティベート時のものの 2 つである. レイヤ数が 2

であることから、表 5 の計算式に当てはめて、ヒープメモリを 48 バイト削減できる。

また、ベースメソッドを空のメソッドにして、レイヤごとと共通の処理を各パーシャルメソッドに記述することにより、`proceed` を使わないで実装することが可能である。この際、レイヤ数 2、クラス数 2、メソッド数 4、インスタンス数 2 であることから、表 5 の計算式に当てはめると、48 バイト削減できる。ただし、ベースメソッドにまとめたプログラムコードが各パーシャルメソッドに散らばるため、プログラムサイズは大きくなる。

6. おわりに

本論文では、C++ベースの組込みソフトウェア向けの COP フレームワーク RTCOP を提案した。提案フレームワークの目的は、組込みソフトウェア開発で使われやすい C++ で COP 実現することである。提案フレームワークは、C++ の COP 拡張のために、メソッドディスパッチの拡張やレイヤアクティベーション機構の提供を行う。また、これらの仕組みが組込みソフトウェアに適用できる性能を持つように、以下の特徴を備える。(1) メモリ消費量を抑えるために構成可能な作りである、(2) メソッドディスパッチ・レイヤアクティベーションの速度が実用的なレベルに収まる。(1) により、COP の各種機能のうち、アプリケーションによっては不要な機能を外すことで、メモリ消費量を節約する。(2) により、性能が求められる組込みソフトウェアへの適用を可能とする。

我々は提案フレームワークの機能と性能が妥当な範囲であることについて述べるために、メソッドディスパッチとレイヤアクティベーションの実行速度を計測し、我々が想定する組込みシステムの要求と比較した。さらに、提案フレームワークの組込みソフトウェアへの適用の優位性を示すために、他の COP 言語と性能比較を行った。結果から、提案フレームワークは要求する性能を満たし、他の COP 言語よりも性能が良いことを示した。

今後は、実際の組込みシステムで用いられるプロセッサ上で提案フレームワークを動作させ、機能的・性能的に妥当であるか評価を行いたい。

謝辞 本研究の一部は科研費（課題番号：15H05708）の助成を受けたものである。

参考文献

- [1] Hirschfeld, R., Costanza, P. and Nierstrasz, O.: Context-oriented Programming, *Journal of Object Technology*, Vol.7, No.3, pp.125–151 (2008).
- [2] Salvaneschi, G., Ghezzi, C. and Pradella, M.: Context-oriented Programming: A Software Engineering Perspective, *Journal of Systems and Software*, Vol.85, No.8, pp.1801–1817 (2012).
- [3] Appeltauer, M., Hirschfeld, R. and Lincke, J.: Declarative Layer Composition with the JCop Programming

- Language, *Journal of Object Technology*, Vol.12, No.4, pp.4:1–37 (2013).
- [4] Appeltauer, M., Hirschfeld, R., Haupt, M. and Masuhara, H.: ContextJ: Context-oriented Programming with Java, *Proc. JSSST Annual Conference 2009*, pp.1–15 (2009).
- [5] Costanza, P. and Hirschfeld, R.: Language Constructs for Context-oriented Programming: An Overview of ContextL, *DLS '05: Proc. 2005 Symposium on Dynamic Languages*, pp.1–10 (2005).
- [6] Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J. and Perscheid, M.: A Comparison of Context-oriented Programming Languages, *Proc. Workshop on Context-oriented Programming (COP 2009)*, pp.1–6 (2009).
- [7] Hirschfeld, R., Costanza, P. and Haupt, M.: An Introduction to Context-Oriented Programming with ContextS, Generative and Transformational Techniques in Software Engineering II (GTTSE), LNCS 5235, pp.396–407, Springer (2008).
- [8] Lincke, J., Appeltauer, M., Steinert, B. and Hirschfeld, R.: An Open Implementation for Context-oriented Layer Composition in ContextJS, *Journal on Science of Computer Programming*, Vol.76, No.12, pp.1194–1209 (2011).
- [9] von Lowis, M., Denker, M. and Nierstrasz, O.: Context-oriented Programming: Beyond Layers, *ICDL '07: Proc. 2007 International Conference on Dynamic Languages*, pp.143–156, ACM (2007).
- [10] 青谷知幸, 紙名哲生, 増原英彦: オブジェクト毎の層遷移を宣言的に記述できる文脈指向言語 EventCJ, コンピュータソフトウェア, Vol.30, No.3, pp.130–147 (2013).
- [11] Rasche, A., Schult, W. and Polze, A.: Self-Adaptive Multithreaded Applications - A Case for Dynamic Aspect Weaving, *ARM '05: Proc. 4th Workshop on Reflective and Adaptive Middleware Systems*, Article No.10 (2005).
- [12] Gonzalez, S., Cardozo, N., Mens, K., Cadiz, A., Libbrecht, J. and Goffaux, J.: Subjective-C: Bringing Context to Mobile Platform Programming, *SLE '10: Proc. 3rd International Conference on Software Language Engineering (SLE 2010)*, pp.246–265 (2010).
- [13] 紙名哲生: 文脈指向プログラミングの要素技術と展望, コンピュータソフトウェア, Vol.31, No.1, pp. 3–13 (2014).
- [14] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An overview of AspectJ, *ECOOP 2001*, LNCS 2072, pp.327–353 (2001).
- [15] Apel, S., Leich, T., Rosenmuller, M. and Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming, *Proc. International Conference on Generative Programming and Component Engineering*, pp.125–140 (2005).
- [16] 佐伯優太, 谷川郁太, 久住憲嗣, 福田 晃: ContextROS: ロボットペレーティングシステムへのコンテキスト指向プログラミングの適用, 組込みシステムシンポジウム 2017 論文集, pp.47–53 (2017).



谷川 郁太 (正会員)

2013 年東海大学情報通信学部組込みソフトウェア工学科卒業。2015 年東海大学大学院情報通信学研究科修士課程修了。現在、九州大学大学院システム情報科学府博士後期課程在学中。コンテキスト指向プログラミングに関する研究に従事。

る研究に従事。



久住 憲嗣 (正会員)

1977 年生まれ。1999 年 3 月信州大学工学部情報工学科卒業。2001 年 3 月奈良先端科学技術大学院大学情報科学研究科博士前期課程修了。2004 年 3 月九州大学大学院システム情報科学府博士後期課程修了。2004 年 4 月～

2005 年 9 月科学技術振興機構研究員。2005 年 10 月～2008 年 6 月九州大学システム LSI 研究センター特任講師。2008 年 7 月から九州大学システム LSI 研究センター准教授。組込みソフトウェア開発方法論の教育、研究に従事。特にモデル駆動開発や低消費電力化技術に関する研究に従事。



小倉 信彦

1997 年東京工業大学大学院理工学研究科修士修了。同年東京工業大学精密工学研究所助手。2005 年武蔵工業大学講師。2007 年 4 月同准教授。校名変更 (2009 年)、改組 (2013 年) にと

もない東京都市大学メディア情報学部情報システム学科准教授。現在に至る。博士 (工学)。信号処理に関する研究に従事。



菅谷 みどり (正会員)

2004 年早稲田大学大学院情報科学科修士課程修了。2007 年同大学院情報ネットワーク専攻修了。2008 年独立行政法人科学技術振興機構戦略的創造研究推進機構 (CREST) デイペンダブル組込み OS 研究開発センター研究

員。2010 年横浜国立大学。未来情報通信医療社会基盤センター講師。2013 年芝浦工業大学工学部准教授。2018 年同大学教授。現在に至る。工学博士。組込みソフトウェア、オペレーティングシステム、ユビキタスコンピューティングに関する研究に従事。電子情報通信学会、ACM、IEEE Computer Society。



渡辺 晴美 (正会員)

1998 年東京工業大学大学院博士課程修了。博士 (工学)。2004 年 4 月東海大学勤務。2011 年東海大学情報通信学部教授。2016 年から情報処理学会組込みシステム研究会主査。ロボット

を利用した教育。組込みソフトウェアの開発方法論に関する研究に従事。特に環境依存型システムのモデル駆動開発に興味を持つ。



福田 晃 (正会員)

1977 年九州大学工学部情報工学科卒業。1979 年同大学大学院工学研究科修士課程情報工学専攻修了。同年日本電信電話公社 (現 NTT) 武蔵野電気通信研究所入所。1983 年九州大学助

手。1989 年同大学助教授。1994 年奈良先端科学技術大学院大学教授。2001 年九州大学大学院システム情報科学研究院教授。2008 年九州大学システム LSI 研究センター長 (兼任)。2015 年九州大学主幹教授。2016 年九州大学スマートモビリティ研究開発センター長 (兼任)。現在に至る。工学博士。組込みソフトウェア、ユビキタスコンピューティングに関する研究に従事。情報処理学会研究賞 (平成 2 年)、Best Author 賞 (平成 5 年) 等を受賞。電子情報通信学会、ACM、IEEE Computer Society、日本 OR 学会各会員、「NPO 法人九州組込みソフトウェアコンソーシアム (QUEST)」理事長、「九州 IoT コミュニティ」会長等。本会フェロー。