

2020年5月20日
ユビキタスロボティクス特論



OpenRTM-aistおよび RTコンポーネントプログラミングの概要

国立研究開発法人産業技術総合研究所
ロボットイノベーション研究センター
ロボットソフトウェアプラットフォーム研究チーム長
安藤 慶昭

はじめに

- RTミドルウェアの概要
 - 基本概念
- ロボットソフトウェアの動向
- モジュール化のメリット
- RTコンポーネントの基本機能
- 標準化
- コミュニティ

RTミドルウェアとは？

RTとは?

- RT = Robot Technology cf. IT
 - ≠Real-time
 - 単体のロボットだけでなく、さまざまなロボット技術に基づく機能要素をも含む (センサ、アクチュエータ, 制御スキーム、アルゴリズム、etc….)

産総研版RTミドルウェア

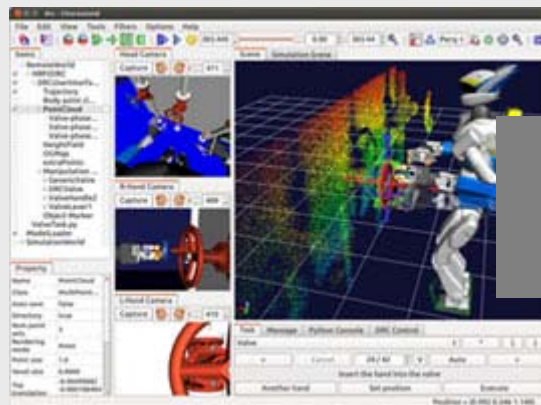
OpenRTM-aist

- RT-Middleware (RTM)
 - RT要素のインテグレーションのためのミドルウェア
- RT-Component (RTC)
 - RT-Middlewareにおけるソフトウェアの基本単位

ロボットミドルウェアについて

- ロボットシステム構築を効率化するための共通機能を提供する**基盤ソフトウェア**
 - － 「ロボットOS」と呼ばれることもある
 - － インターフェース・プロトコルの共通化、標準化
 - － 例として
 - モジュール化・コンポーネント化フレームワークを提供
 - モジュール間の通信をサポート
 - パラメータの設定、配置、起動、モジュールの複合化（結合）機能を提供
 - 抽象化により、OSや言語間連携・相互運用を実現
- 2000年ごろから開発が活発化
 - － 世界各国で様々なミドルウェアが開発・公開されている

従来のシステムでは…



Controller

Controller
software



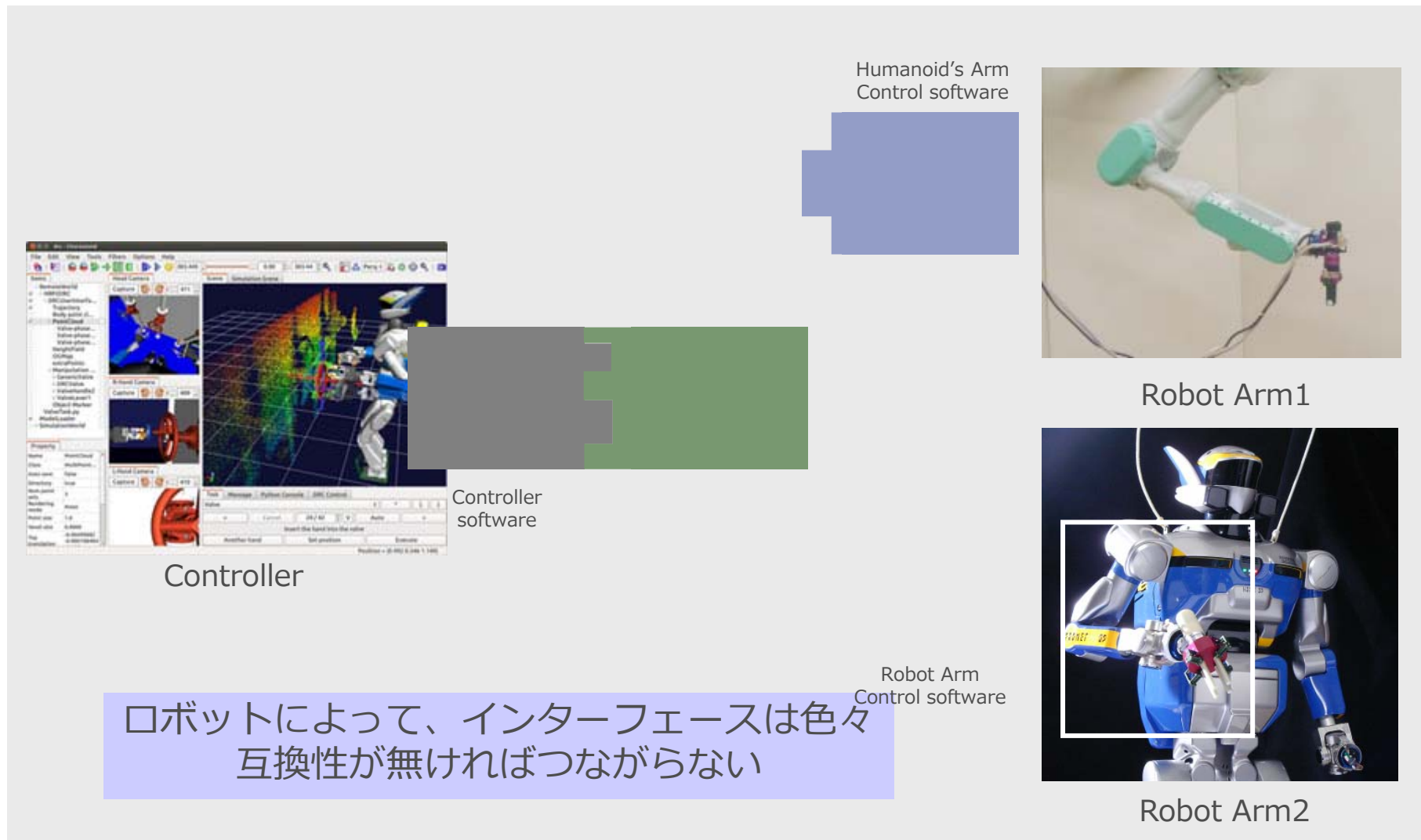
Robot Arm
Control software



Robot Arm2

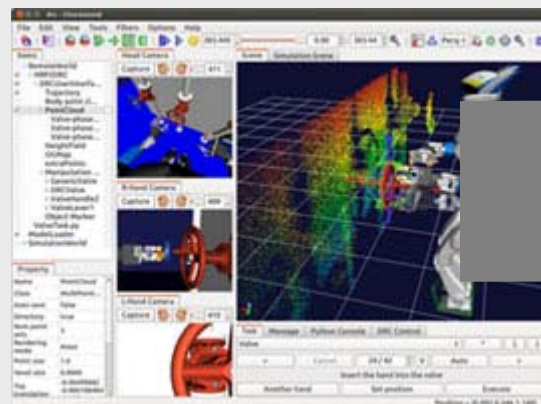
互換性のあるインターフェース同士は接続可能

従来のシステムでは…



RTミドルウェアでは…

RTミドルウェアは別々に作られたソフトウェアモジュール同士を繋ぐための共通インターフェースを提供する



Controller

Controller software

Arm A
Control software



Robot Arm1

compatible
arm interfaces

Arm B
Control software



Robot Arm2

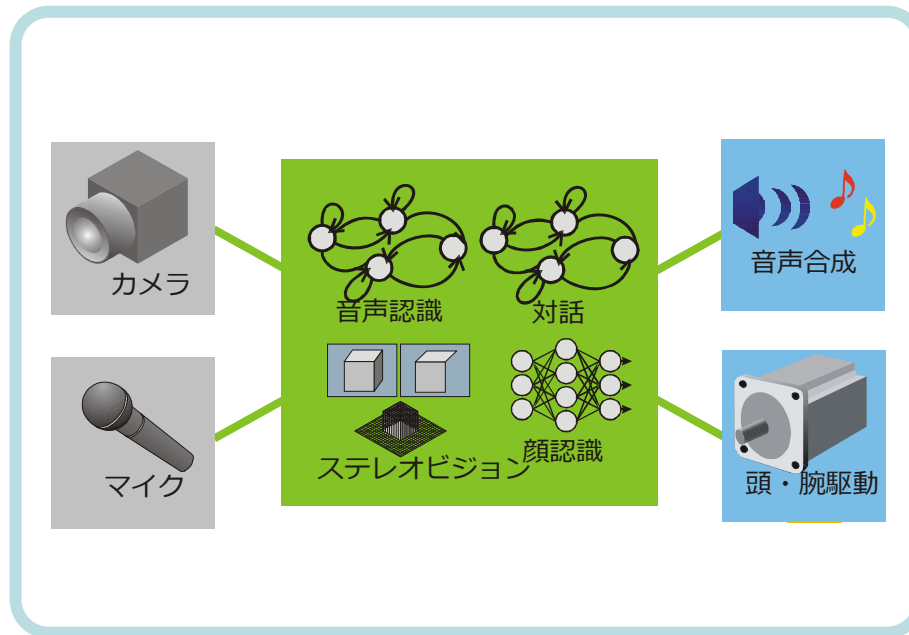
ソフトウェアの再利用性の向上
RTシステム構築が容易になる

ロボットソフトウェア開発の方向

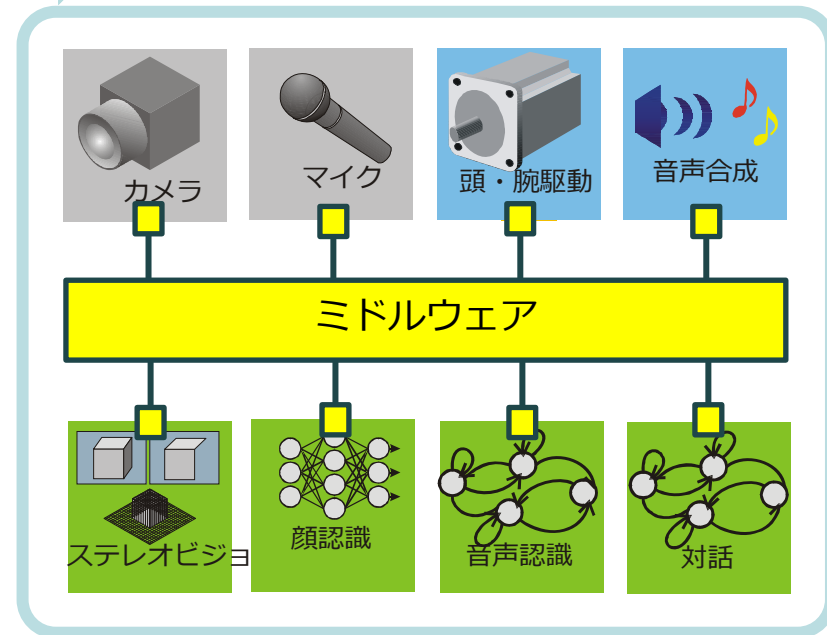
従来型開発



コンポーネント指向開発



- ✓ 様々な機能を融合的に設計
- ✓ 実行時の効率が高いが、柔軟性に欠ける
- ✓ システムが複雑化してくると開発が困難に



- ✓ 大規模複雑な機能の分割・統合
- ✓ 開発・保守効率化（機能の再利用等）
- ✓ システムの柔軟性向上

モジュール化のメリット

- 再利用性の向上
 - 同じコンポーネントをいろいろなシステムに使いまわせる
- 選択肢の多様化
 - 同じ機能を持つ複数のモジュールを試すことができる
- 柔軟性の向上
 - モジュール接続構成かえるだけで様々なシステムを構築できる
- 信頼性の向上
 - モジュール単位でテスト可能なため信頼性が向上する
- 堅牢性の向上
 - システムがモジュールで分割されているので、一つの問題が全体に波及しにくい

RTコンポーネント化のメリット

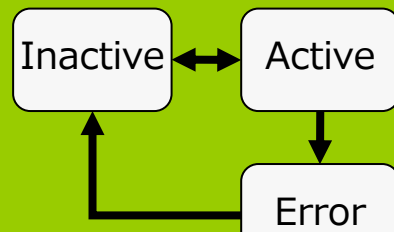
モジュール化のメリットに加えて

- ソフトウェアパターンを提供
 - ロボットに特有のソフトウェアパターンを提供することで、体系的なシステム構築が可能
- フレームワークの提供
 - フレームワークが提供されているので、コアのロジックに集中できる
- 分散ミドルウェア
 - ロボット体内LANやネットワークロボットなど、分散システムを容易に構築可能

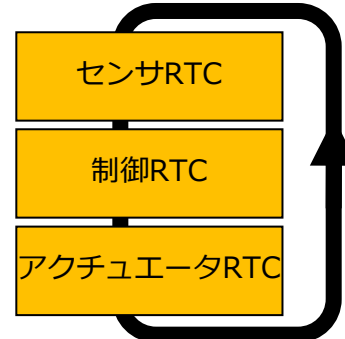
RTコンポーネントの主な機能

アクティビティ・実行コンテキスト

共通の状態遷移



複合実行

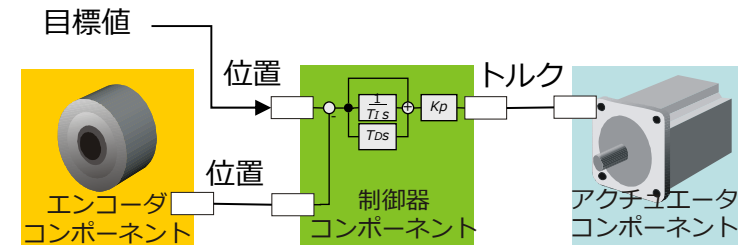


ライフサイクルの管理・コアロジックの実行

データポート

- データ指向ポート
- 連続的なデータの送受信
- 動的な接続・切断

サーボの例



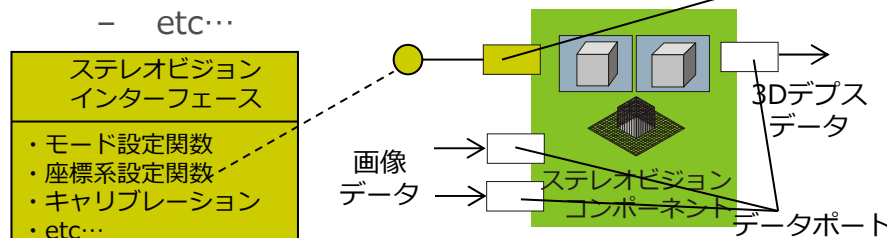
データ指向通信機能

サービスポート

- 定義可能なインターフェースを持つ
- 内部の詳細な機能にアクセス
 - パラメータ取得・設定
 - モード切替
 - etc...

ステレオビジョンの例

サービスポート

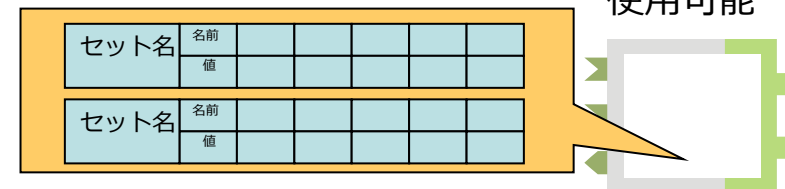


サービス指向相互作用機能

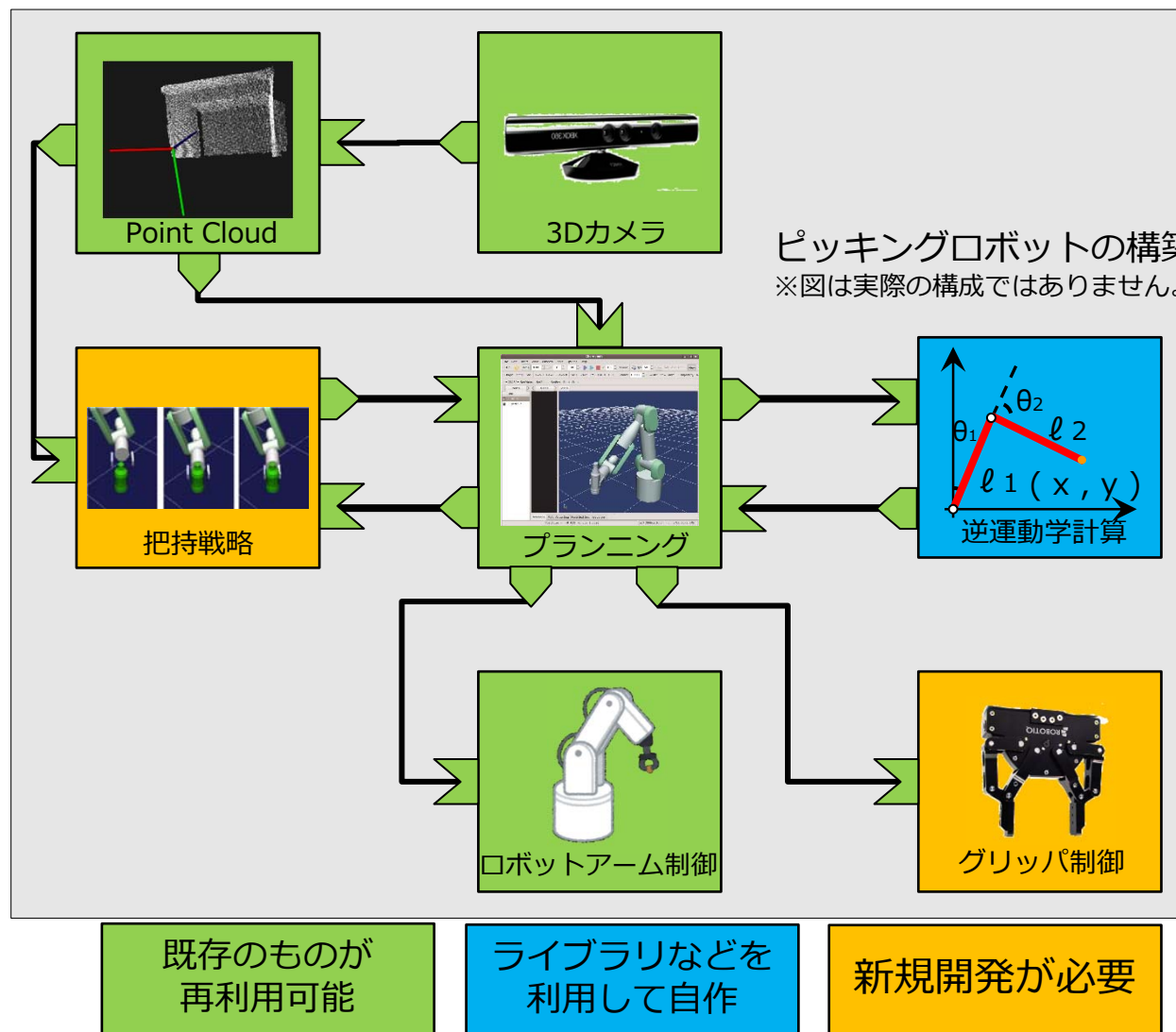
コンフィギュレーション

- パラメータを保持する仕組み
- いくつかのセットを保持可能
- 実行時に動的に変更可能

複数のセットを
動作時に
切り替えて
使用可能



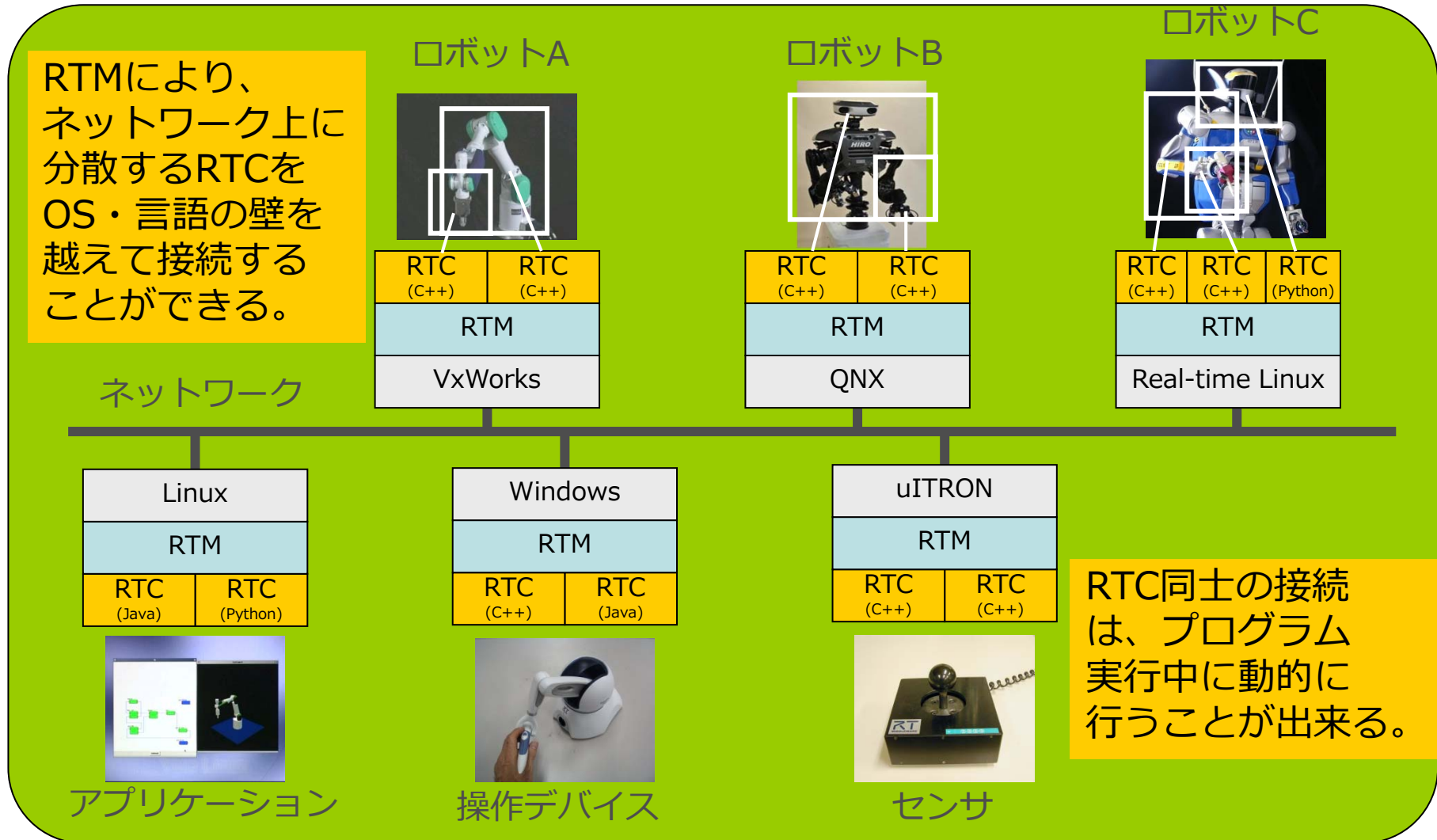
ミドルウェアを利用した開発の利点



ミドルウェアを利用すると、**既存のモジュール**が利用できる

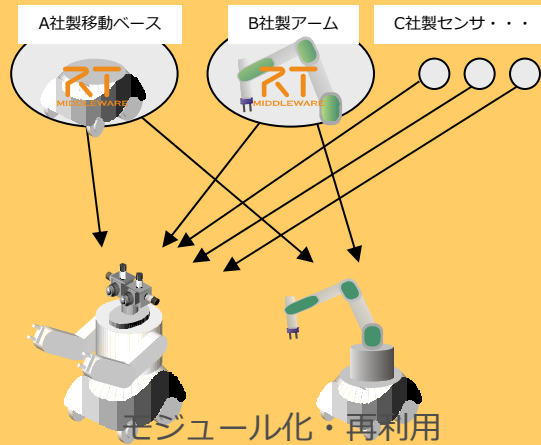
開発するときに**新規に作らなければならない部分**は少なくて済む

RTミドルウェアによる分散システム



RTミドルウェアの目的 モジュール化による問題解決

コストの問題



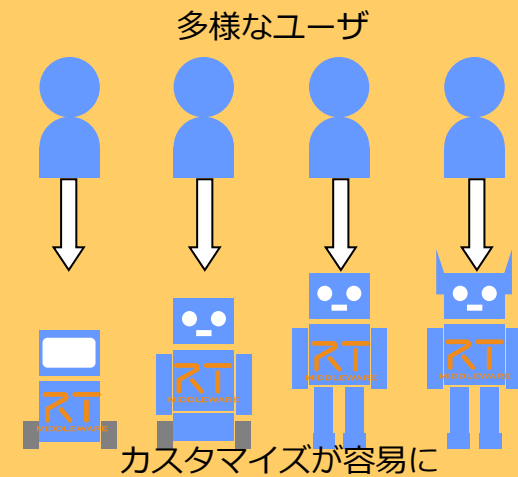
ロボットの低コスト化

技術の問題



最新技術を利用可能

ニーズの問題



多様なニーズに対応

ロボットシステムインテグレーションによるイノベーション

実用例・製品化例



HRPシリーズ: 川田工業、AIST

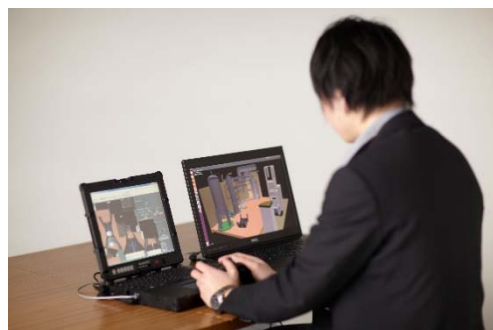


S-ONE : SCHAFT



DAQ-Middleware: KEK/J-PARC

KEK: High Energy Accelerator Research Organization
J-PARC: Japan Proton Accelerator Research Complex



災害対応ロボット操縦シミュレータ:
NEDO/千葉工大



HIRO, NEXTAGE open: Kawada Robotics



RAPUDA : Life Robotics



ビュートロバーRTC/RTC-BT(VSTONE)



OROCHI (アールティ)



新日本電工他: Mobile SEM

RTミドルウェアは国際標準

Date: September 2012



Robotic Technology Component (RTC)

Version 1.1

Normative reference: <http://www.omg.org/spec/RTC/1.1>
 Machine consumable files: <http://www.omg.org/spec/RTC/20111205/>
 Normative:
<http://www.omg.org/spec/RTC/20111205/rtc.xml>
<http://www.omg.org/spec/RTC/20111205/rtc.h>
<http://www.omg.org/spec/RTC/20111205/rtc.idl>
 Non-normative:
<http://www.omg.org/spec/RTC/20111205/rtc.eap>

標準化履歴

- 2005年9月
Request for Proposal 発行(標準化開始)
- 2006年9月
OMGで承認、事実上の国際標準獲得
- 2008年4月
OMG RTC標準仕様 ver.1.0公式リリース
- 2012年9月
ver. 1.1改定
- 2015年9月
FSM4RTC(FSM型RTCとデータポート標準) 採択

OMG国際標準

標準化組織で手続きに沿って策定

- 1組織では勝手に改変できない安心感
- 多くの互換実装ができつつある
- 競争と相互運用性が促進される

RTミドルウェア互換実装は10種類以上

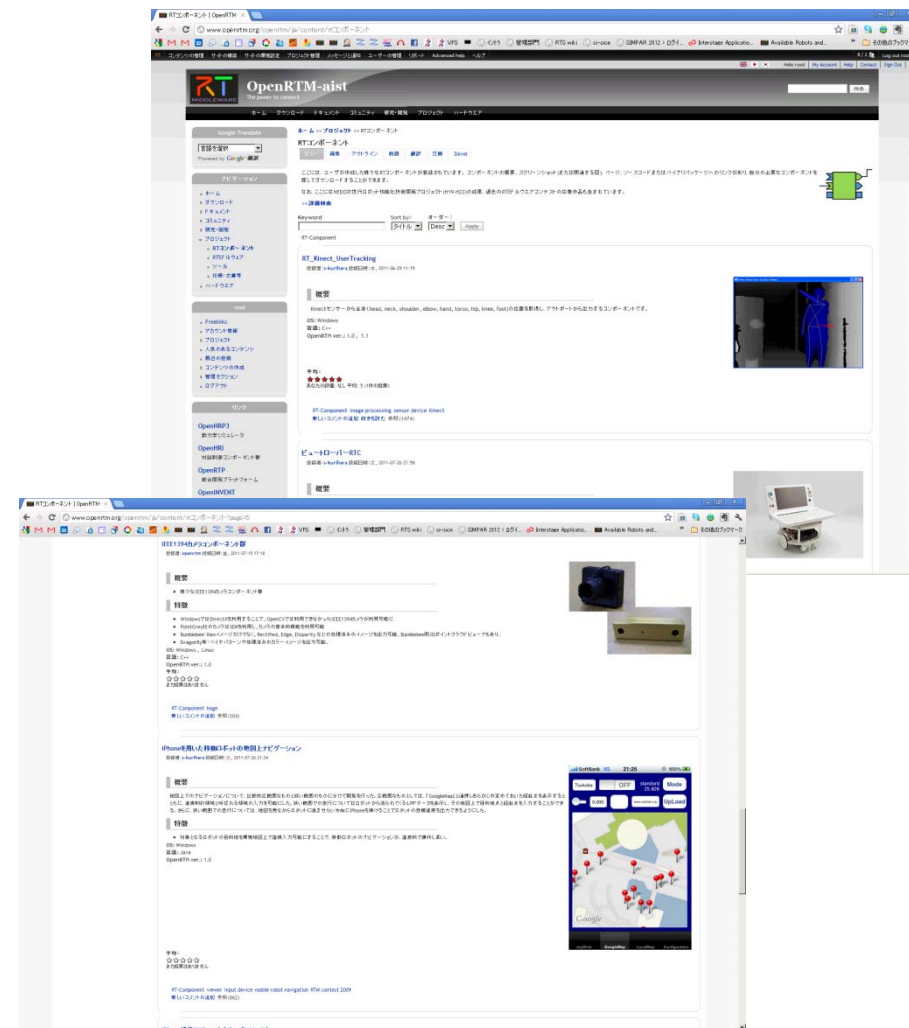
| 名称 | ベンダ | 特徴 | 互換性 |
|----------------|----------------|----------------------------------|-----|
| OpenRTM-aist | 産総研 | NEDO PJで開発。参照実装。 | --- |
| HRTM | ホンダ | アシモはHRTMへ移行中 | ◎ |
| OpenRTM.NET | セック | .NET(C#, VB, C++/CLI, F#, etc..) | ◎ |
| RTM on Android | セック | Android版RTミドルウェア | ◎ |
| RTC-Lite | 産総研 | PIC, dsPIC上の実装 | ○ |
| Mini/MicorRTC | SEC | NEDOオープンイノベーションPJで開発 | ○ |
| RTMSafety | SEC/AIST | NEDO知能化PJで開発・機能安全認証取得 | ○ |
| RTC CANOpen | SIT, CiA | CAN業界RTM標準 | ○ |
| PALRO | 富士ソフト | 小型ヒューマノイドのためのC++ PSM 実装 | × |
| OPRoS | ETRI | 韓国国家プロジェクトでの実装 | × |
| GostaiRTC | GOSTAI, THALES | ロボット言語上で動作するC++ PSM 実装 | × |

特定のベンダが撤退しても
ユーザは使い続けることが可能

プロジェクトページ

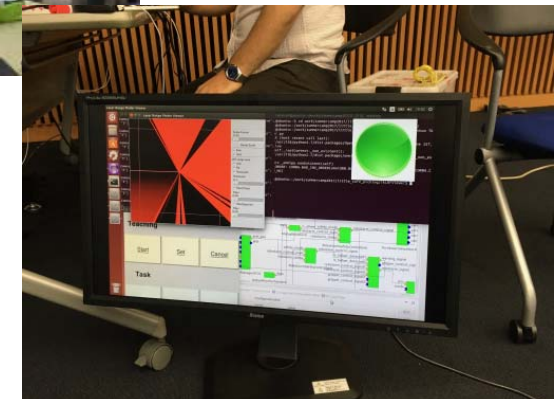
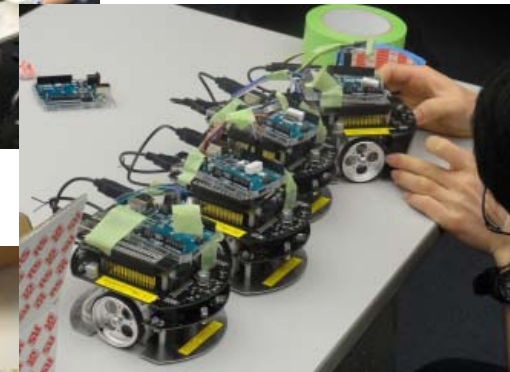
- ユーザが自分の作品を登録
- 他のユーザの作ったRTCを探ることができる

| タイプ | 登録数 |
|------------|-----|
| RTコンポーネント群 | 287 |
| RTミドルウェア | 14 |
| ツール | 19 |
| 仕様・文書 | 4 |
| ハードウェア | 28 |



サマーキャンプ

- 毎年夏に1週間開催
- 今年：7月30日～8月3日
- 募集人数：20名
- 場所：産総研つくばセンター
- 座学と実習を1週間行い、最後にそれぞれが成果を発表
- 産総研内のさくら館に宿泊しながら夜通し？コーディングを行う！



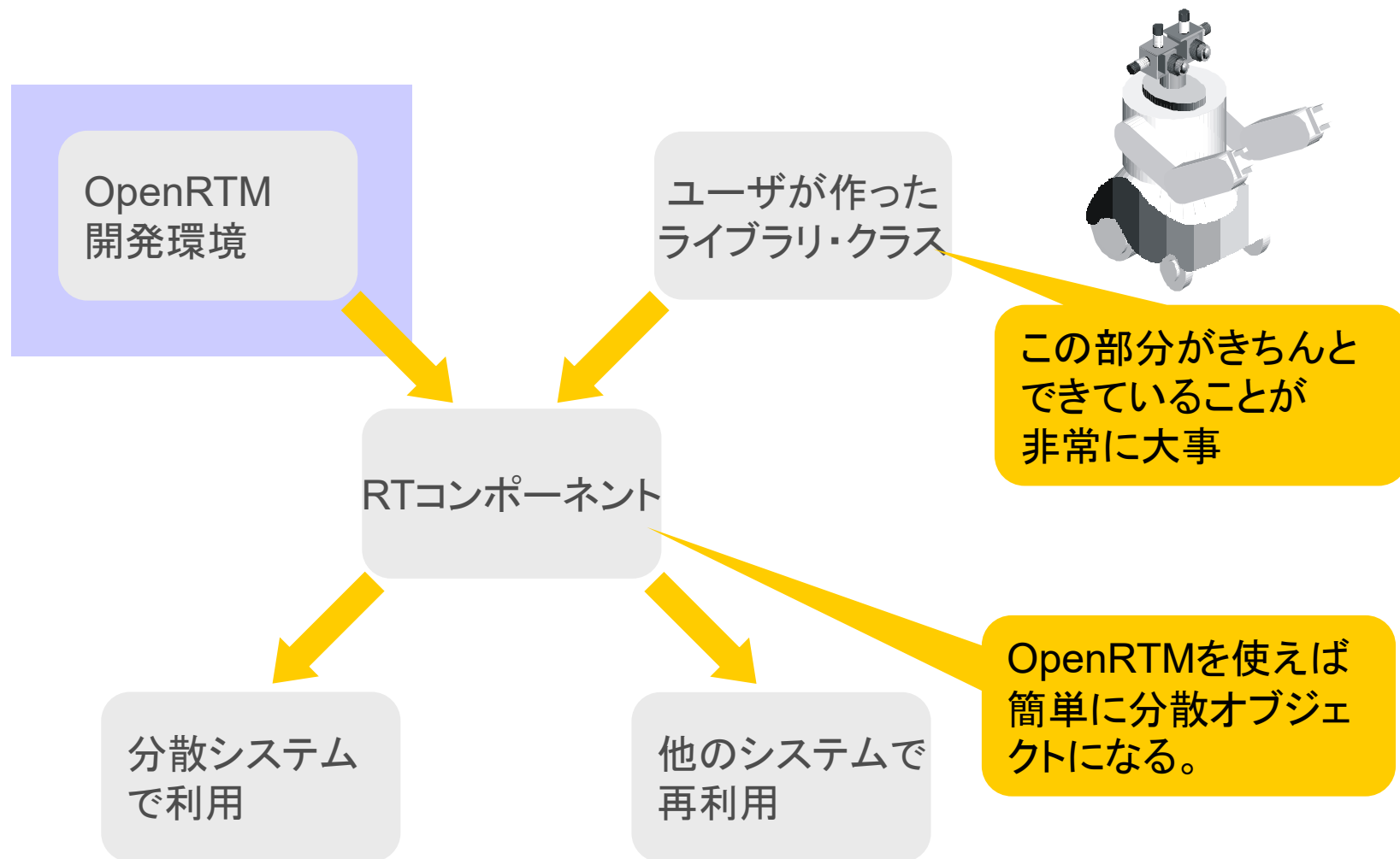
RTミドルウェアコンテスト

- SICE SI（計測自動制御学会 システムインテグレーション部門講演会）のセッションとして開催
 - 各種奨励賞・審査基準開示:5月頃
 - エントリー〆切：8月10日(SI2018締切)
 - 講演原稿〆切：9月17日
 - ソフトウェア登録：10月ごろ
 - オンライン審査：11月下旬～
 - 発表・授賞式：12月ごろ
- 2017年度実績
 - 応募数：10件
 - 計測自動制御学会学会RTミドルウェア賞（副賞10万円）
 - 奨励賞（賞品協賛）：2件
 - 奨励賞（団体協賛）：9件
 - 奨励賞（個人協賛）：8件
- 詳細はWebページ：openrtm.org
 - コミュニティ→イベント をご覧ください

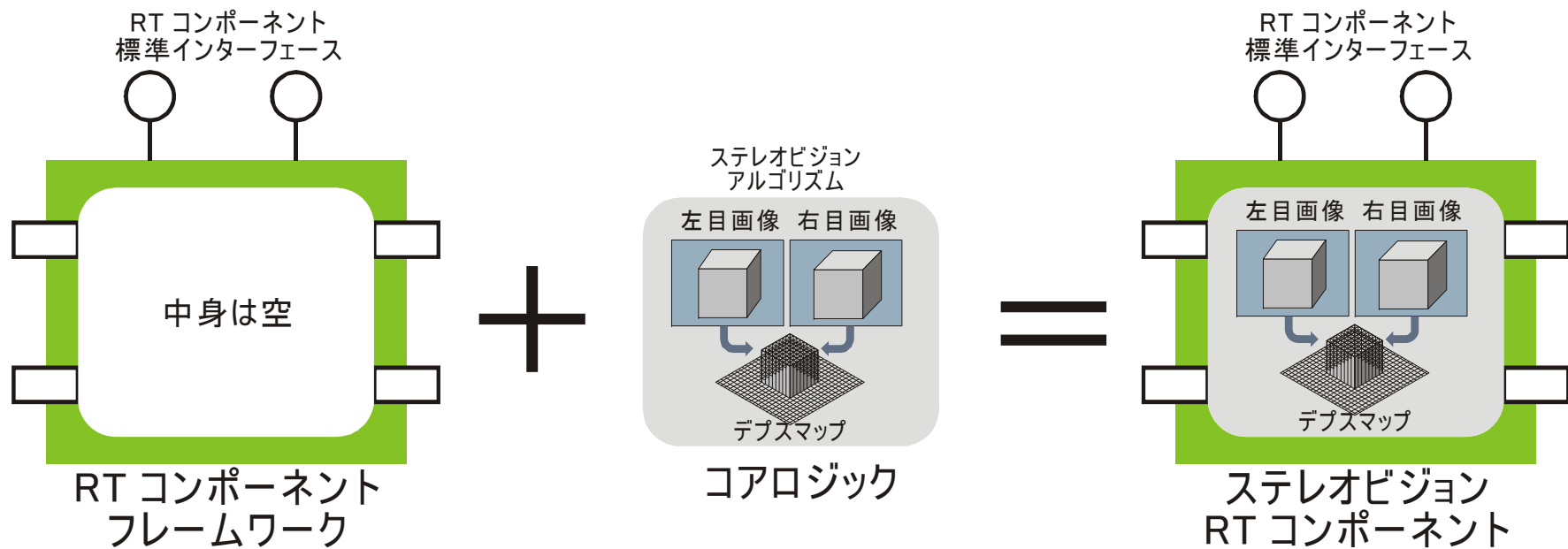


RTC開発の実際

OpenRTMを使った開発の流れ

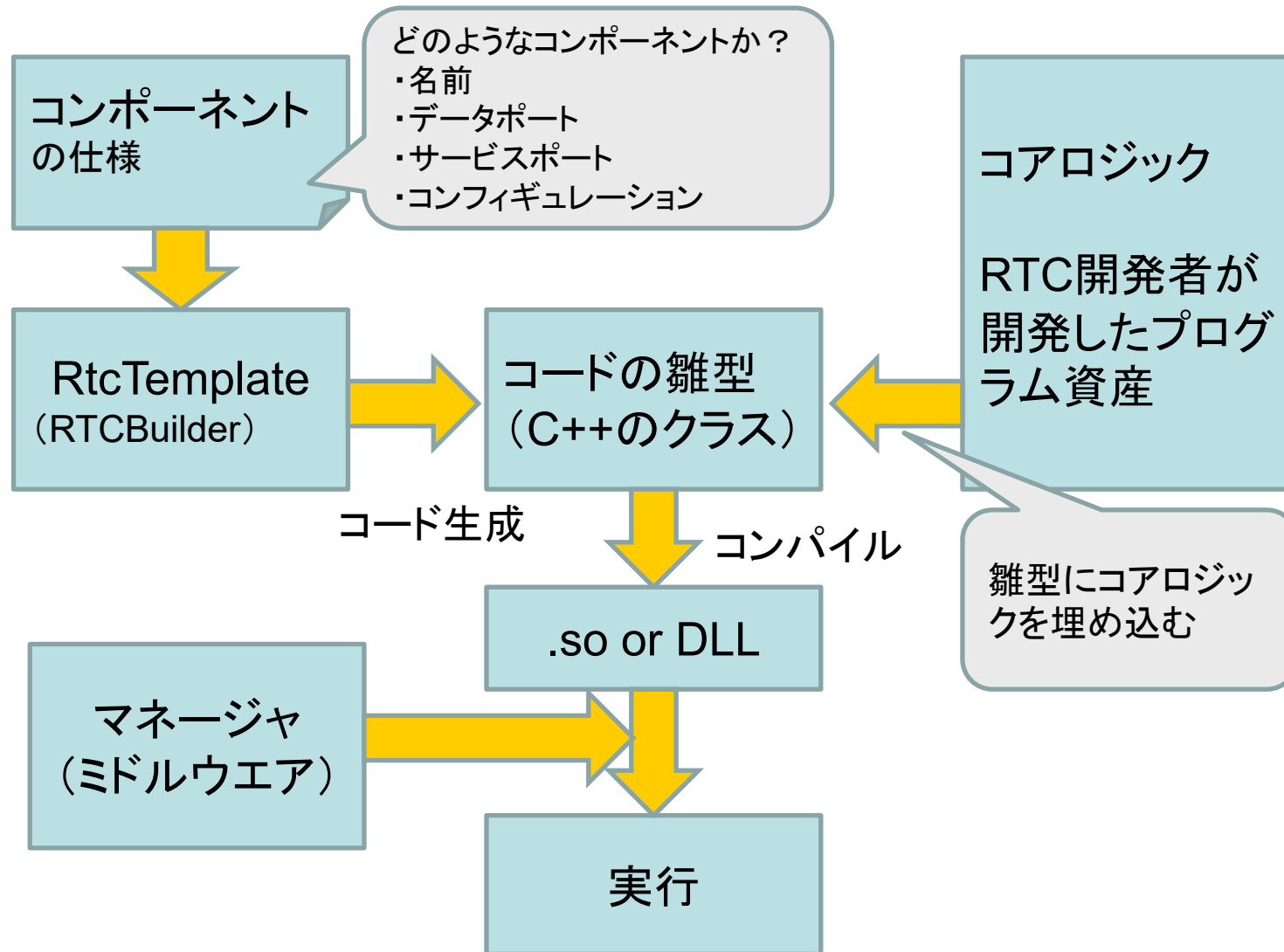


フレームワークとコアロジック



RTCフレームワーク+コアロジック=RTコンポーネント

OpenRTMを使った開発の流れ



モデルに基づくコード生成

コンポーネント仕様

```

name:      MyComp
category:  temp.sensor device
description: temp. sensor RTC
comp_type: STATIC
act_type:  PERIODIC
InPorts:   mode:TimedBool
OutPorts:  temp: TimedDouble
    
```

同一のRTC仕様からは
言語が異なっても、
同じ(コンポーネントモデルの)RTCが生成される

Template code generator

C++
backend

Java
backend

Python
backend

RTC-Lite
backend

RTC source for C++

```

class MyComp
: public DataflowComponent {
public:
    virtual onExecute(ec_id);
    :
private:
    TimedBool m_mode;
    TimedDouble m_temp;
};
    
```

RTC source for Java

```

import RTC.DataFlowComponent;
public class MyCompImpl
extends DataFlowComponent
{
    public ConsoleImpl(mgr)
    {
    }
    :
};
    
```

RTC source for Python

```

#!/usr/bin/env python
import RTC
class MyComp(
    DataFlowComponent):
    def __init__(self, manager):
    :
    def onExecute(self, ec_id):
    :
    
```

RTC-Lite source for PIC C

```

#include <16f877a.h>
#include "rtc_base.c"

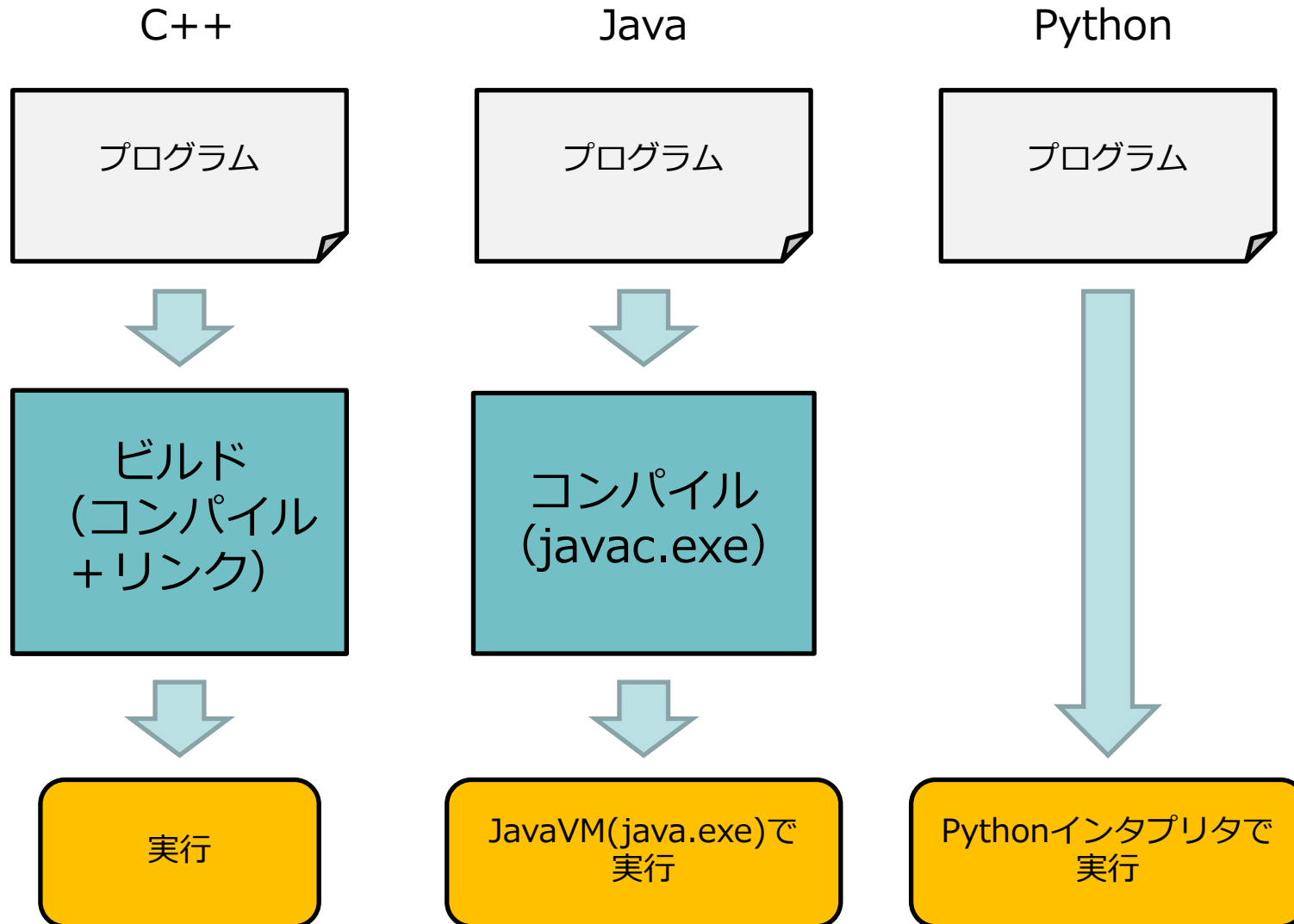
int main (void)
{
    rtc_connect_proxy();
    rtc_mainloop();
    return 0;
}
    
```

RTC-Lite proxy code

```

#!/usr/bin/env python
import RTC
class Proxy(
    DataFlowComponent):
    def __init__(self, manager):
    :
    def onExecute(self, ec_id):
    :
    
```

プログラミングの流れ



CMake

- コンパイラに依存しないビルド自動化のためのフリーソフトウェア
- 様々なOS上の様々な開発環境用ビルドファイルを生成することができる
 - Linux では Makefileを生成
 - Windows ではVC(Visual C++)のプロジェクトファイルを生成
- 最近のオープンソースソフトウェアではCMakeでビルドするようになっているものが多数。

コンポーネント作成の流れ

Windows

RTBuilder

CMake

Visual C++

Linux

RTBuilder

CMake

make
+
gcc (g++)

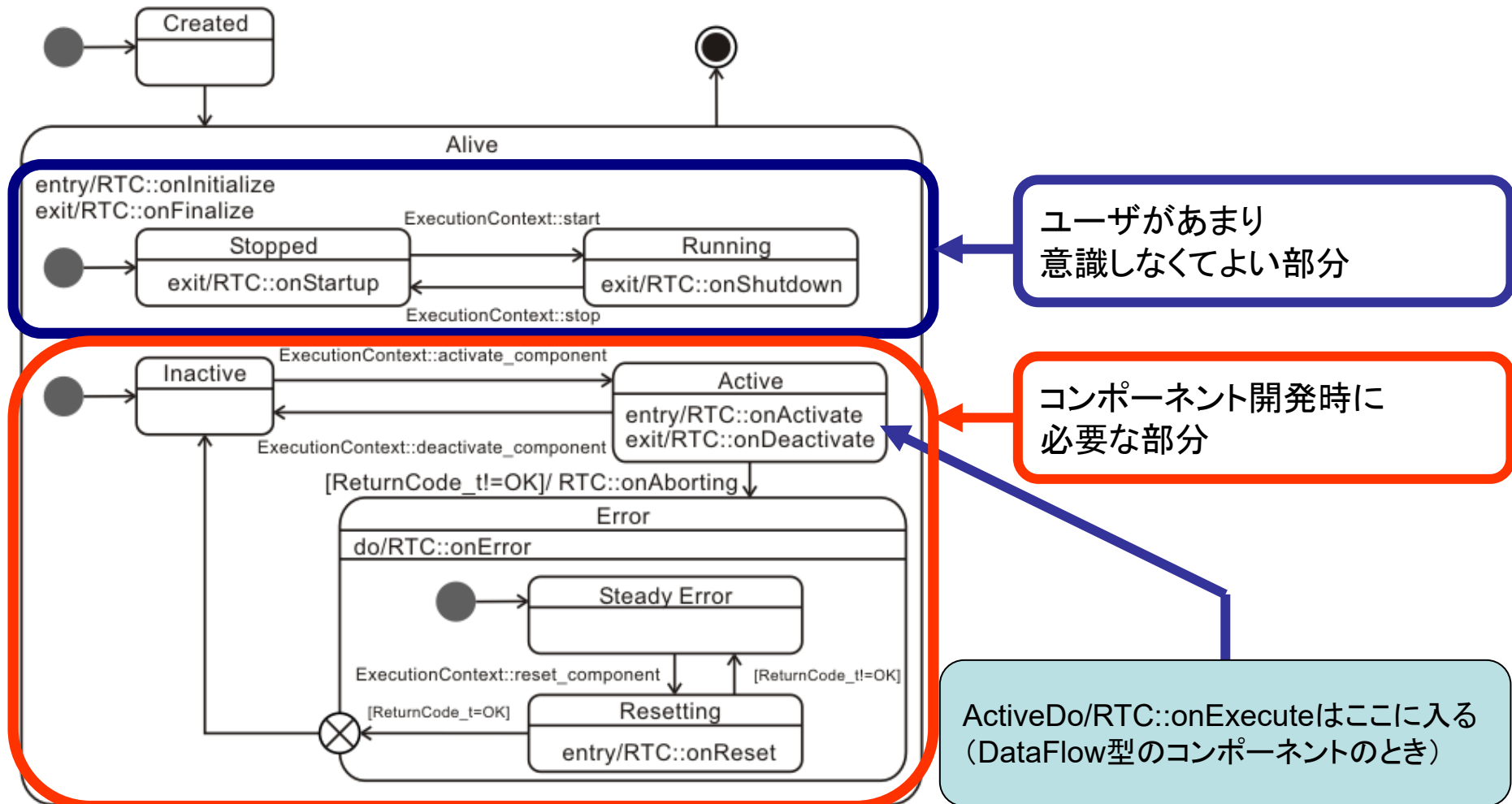
コンポーネントの
仕様の入力

VCプロジェクトファイル
またはMakefileの生成

実装およびコンパイル
実行ファイルの生成

途中まで流れは同じ、コンパイラが異なる

コンポーネント内の状態遷移



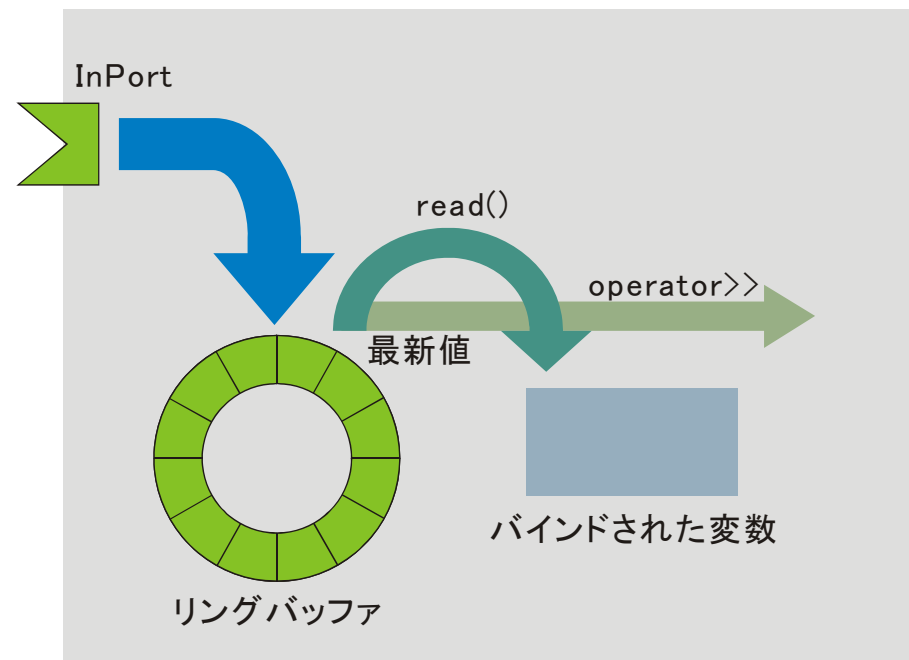
コールバック

- コールバック
 - OutPort
 - バッファ
 - 送受信
 - InPort
- 赤字の部分は、ConnectorListener として新設(右表)
 - 上部(青)は引数にコネクタプロファイルとデータを取るファンクタ
 - 下部(緑)は引数にコネクタプロファイルのみを取るファンクタ

| ラベル | 意味 |
|-------------------------|--------------------|
| ON_BUFFER_WRITE | バッファ書き込み時 |
| ON_BUFFER_FULL | バッファフル時 |
| ON_BUFFER_WRITE_TIMEOUT | バッファ書き込みタイムアウト時 |
| ON_BUFFER_OVERWRITE | バッファ上書き時 |
| ON_BUFFER_READ | バッファ読み出し時 |
| ON_SEND | InProtへの送信時 |
| ON_RECEIVED | InProtへの送信完了時 |
| ON_RECEIVER_FULL | InProt側バッファフル時 |
| ON_RECEIVER_TIMEOUT | InProt側バッファタイムアウト時 |
| ON_RECEIVER_ERROR | InProt側エラー時 |
| ON_BUFFER_EMPTY | バッファが空の場合 |
| ON_BUFFER_READTIMEOUT | バッファが空でタイムアウトした場合 |
| ON_SENDER_EMPTY | OutPort側バッファが空 |
| ON_SENDER_TIMEOUT | OutPort側タイムアウト時 |
| ON_SENDER_ERROR | OutPort側エラー時 |

InPort

- InPortのテンプレート第2引数: バッファ
 - ユーザ定義のバッファが利用可能
- InPortのメソッド
 - read(): InPort バッファからバインドされた変数へ最新値を読み込む
 - >>: ある変数へ最新値を読み込む

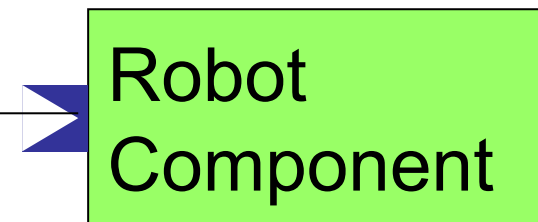


基本的にOutPortと対になる

データポートの型を
同じにする必要あり

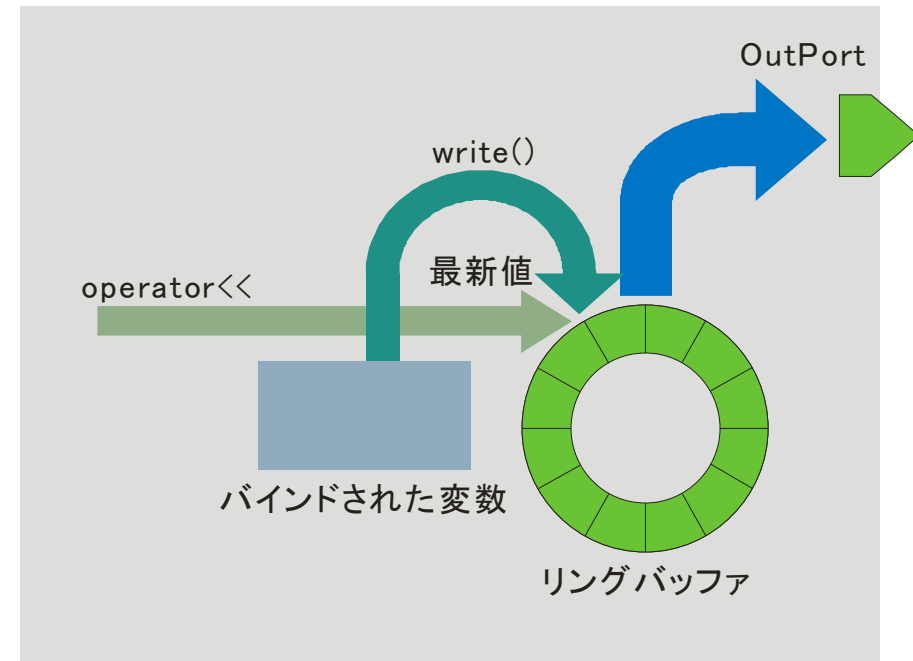
例

Sensor Data

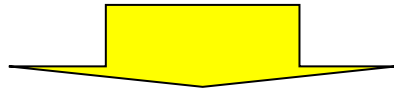


OutPort

- OutPortのテンプレート第2引数:
バッファ
 - ユーザ定義のバッファが利用
可能
- OutPortのメソッド
 - write(): OutPort バッファへ
バインドされた変数の最新値
として書き込む
 - >>: ある変数の内容を最新
値としてリングバッファに書き
込む

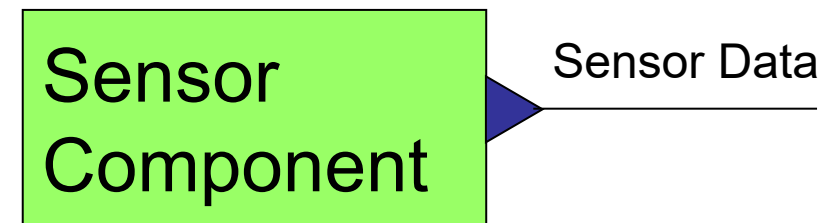


基本的にInPortと対になる



データポートの型を
同じにする必要あり

例



データ変数

```
struct TimedShort
{
    Time tm;
    short data;
};
```

- 基本型
 - tm: 時刻
 - data: データそのもの

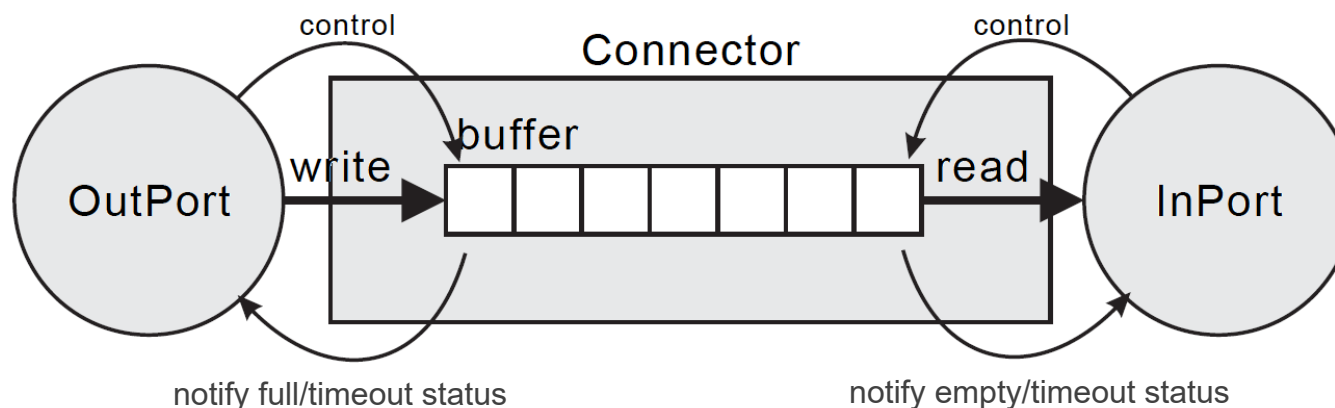
0.2.0では自動で現在時刻を
セットしていたが、0.4.0では
必要に応じて、手動でセット
する必要あり

```
struct TimedShortSeq
{
    Time tm;
    sequence<short> data;
};
```

- シーケンス型
 - data[i]: 添え字によるアクセス
 - data.length(i): 長さiを確保
 - data.length(): 長さを取得
- データを入れるときにはあらかじめ長さをセットしなければならない。
- CORBAのシーケンス型そのもの
- 今後変更される可能性あり

データポートモデル

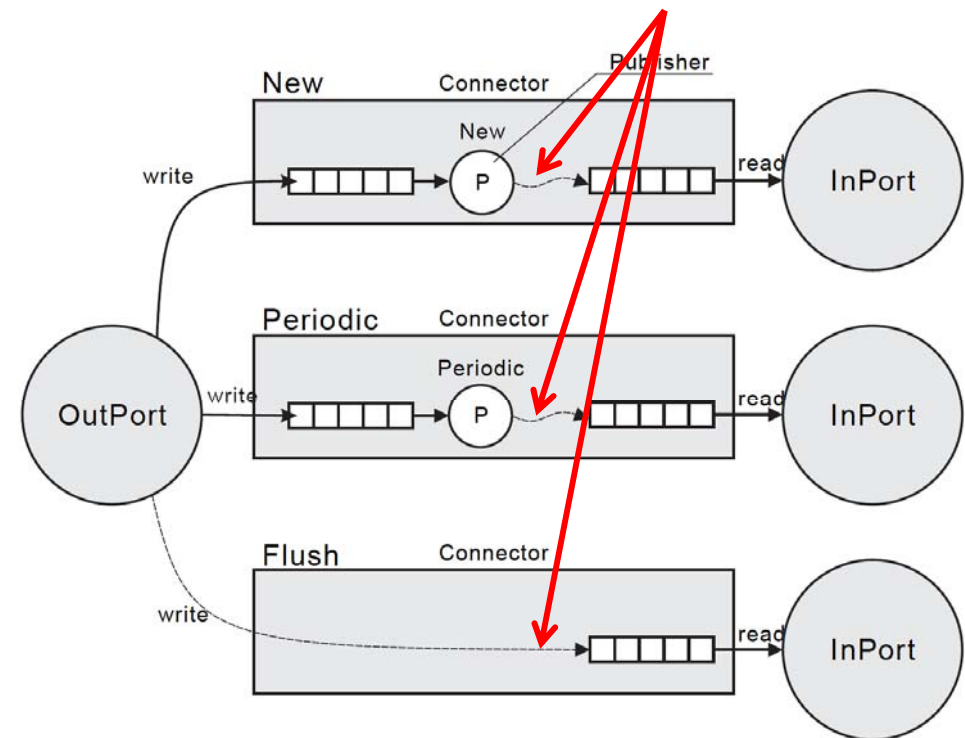
- Connector:
 - バッファと通信路を抽象化したオブジェクト。OutPortからデータを受け取りバッファに書き込む。InPortからの要求に従いバッファからデータを取り出す。
 - OutPortに対してバッファフル・タイムアウト等のステータスを伝える。
 - InPortに対してバッファエンプティ・タイムアウト等のステータスを伝える。
- OutPort:
 - アクティビティからの要求によってデータをコネクタに書き込むオブジェクト
- InPort:
 - アクティビティからの要求によってデータをコネクタから読み出すオブジェクト



Push型データポートモデル

- Connector
 - 実際には間に通信が入る可能性がある
- 3つの送信モデル
 - “new”, “periodic”, “flush”
 - パブリッシャによる実現
- バッファ、パブリッシャ、通信インターフェースの3つをConnectorに内包

ネットワーク等による通信



Pushポリシー

- バッファ残留データ
– 送り方のポリシー

| ポリシー | 送り方 |
|------|---------------|
| ALL | 全部送信 |
| FIFO | 先入れ後だしで1個ずつ送信 |
| NEW | 最新値のみ送信 |
| SKIP | n個おきに間引いて送信 |

- データ生成・消費速度を考慮して設定する必要がある。

予稿にはNEWの説明にLIFOと記述していましたが正確にはLIFOではなく最新値のみの送信です。LIFO形式のポリシーを導入するかどうかは検討中です。ご意見ください。

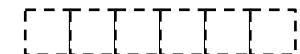
ALL



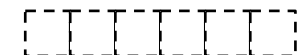
FIFO



NEW



SKIP



コールバック

- コールバック
 - OutPort
 - バッファ
 - 送受信
 - InPort
- 赤字の部分は、ConnectorListener として新設(右表)
 - 上部(青)は引数にコネクタプロファイルとデータを取るファンクタ
 - 下部(緑)は引数にコネクタプロファイルのみを取るファンクタ

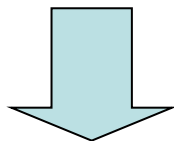
| ラベル | 意味 |
|-------------------------|--------------------|
| ON_BUFFER_WRITE | バッファ書き込み時 |
| ON_BUFFER_FULL | バッファフル時 |
| ON_BUFFER_WRITE_TIMEOUT | バッファ書き込みタイムアウト時 |
| ON_BUFFER_OVERWRITE | バッファ上書き時 |
| ON_BUFFER_READ | バッファ読み出し時 |
| ON_SEND | InProtへの送信時 |
| ON_RECEIVED | InProtへの送信完了時 |
| ON_RECEIVER_FULL | InProt側バッファフル時 |
| ON_RECEIVER_TIMEOUT | InProt側バッファタイムアウト時 |
| ON_RECEIVER_ERROR | InProt側エラー時 |
| ON_BUFFER_EMPTY | バッファが空の場合 |
| ON_BUFFER_READTIMEOUT | バッファが空でタイムアウトした場合 |
| ON_SENDER_EMPTY | OutPort側バッファが空 |
| ON_SENDER_TIMEOUT | OutPort側タイムアウト時 |
| ON_SENDER_ERROR | OutPort側エラー時 |

サービスポート

用途: 他のコンポーネントの提供するサービスを必要なときだけ利用したい時

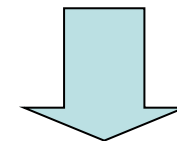
例: カメラからの画像の取得

サービス用のインターフェース(IDL)を定義する



プロバイダ

- インターフェースの実装
- コンポーネントに組み込む(宣言)
- Portへのバインド

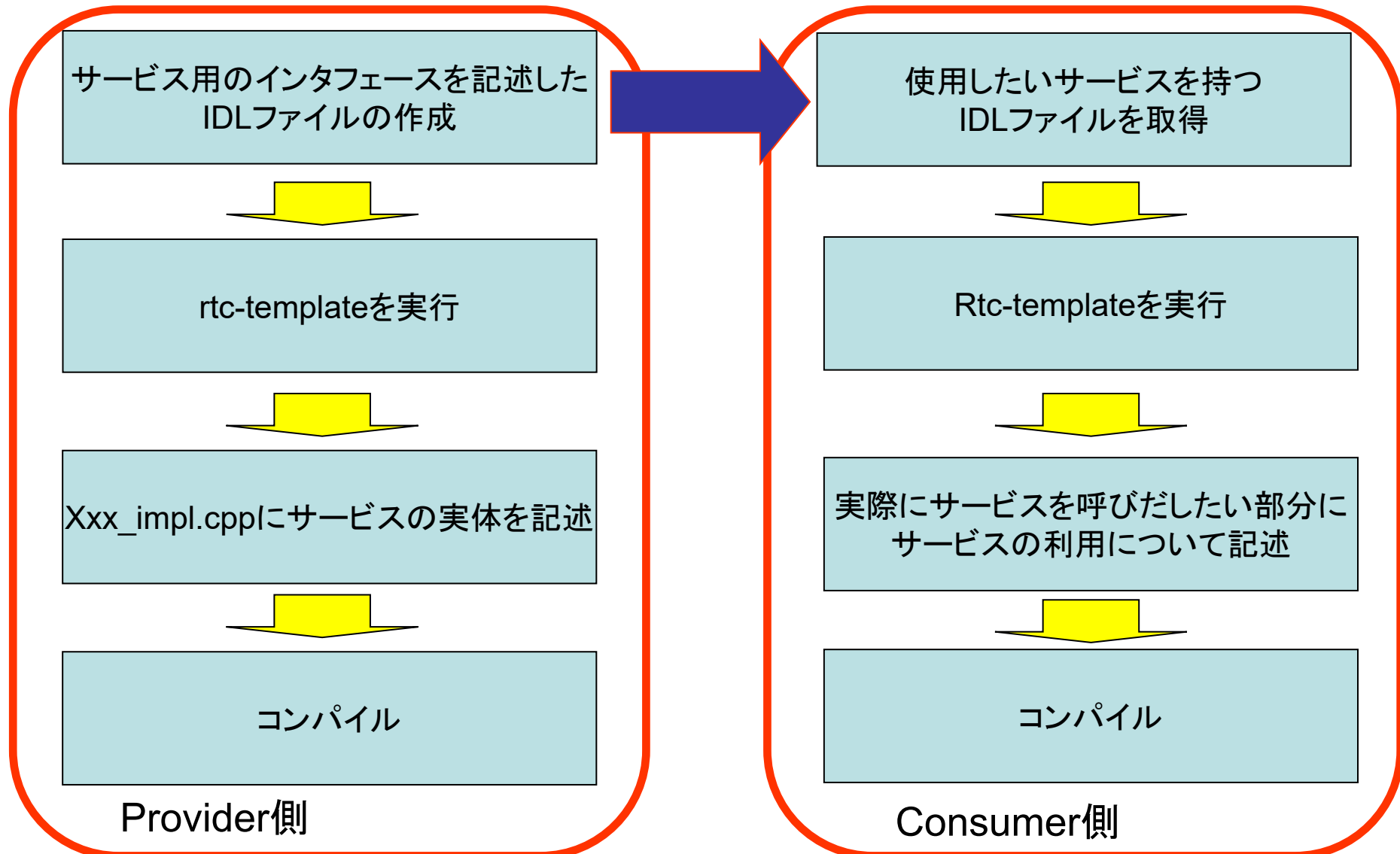


コンシューマ

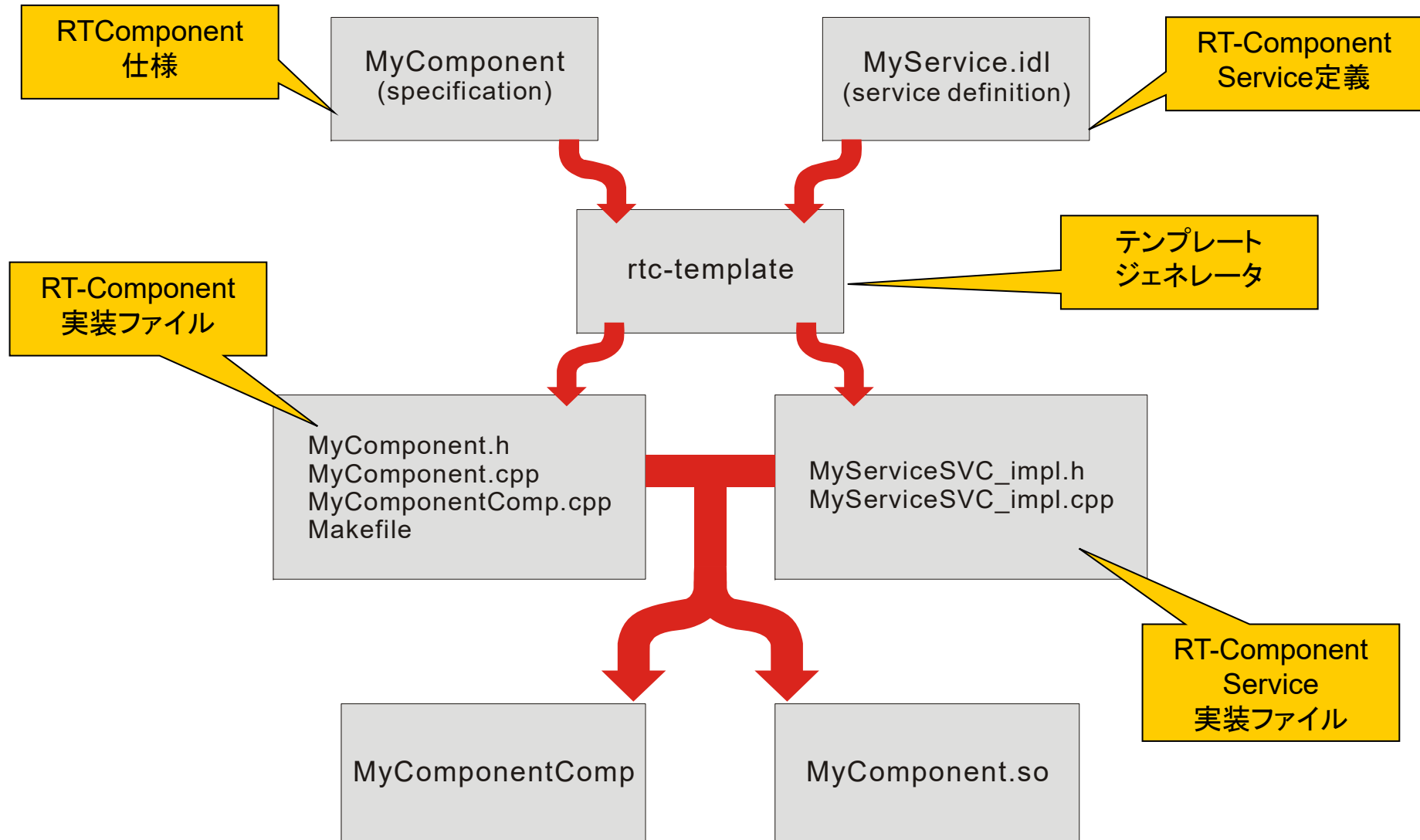
- スタブを組み込む
- コンシューマを宣言
- Portへのバインド

rtc-templateでコンポーネントを作れば上記のことをほとんど気にしないで作れる

サービスポートを使うためのステップ



生成されるファイル



IDLと実装

IDL(CORBA)定義

```
interface MyRobot
{
    // ゲインをセットする
    void setPosCtrlGain(in short axis, in double gain);
    // ゲインを取得する
    double getPosCtrlGain(in int axis);
};
```

サービス実装(雛形はrtc-templateによる自動生成)

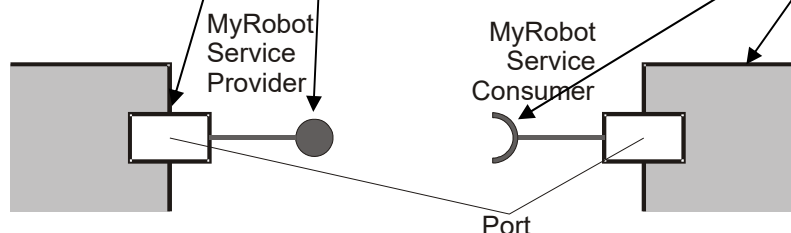
```
class MyRobot_impl
{
    /* この例ではm_robo はロボットを実際に制御する
     * クラスのインスタンスであると仮定する.
     */
    void setPosCtrlGain(const int axis, const double gain)
    {
        // 位置制御ゲインを設定
        m_robo.set_pos_ctrl_gain(axis, gain);
    }
    /* 中略*/
};
```

サービスプロバイダ・コンシューマ

```
class MyRoboComponent
{
private:
    // MyRobot サービスのポートを宣言
    RTC::CorbaPort m_port;
    // MyRobot サービスのインスタンスを宣言
    MyRobot_impl m_robot;
public:
    ManipulatorComponent(Manager manager)
    {
        // ポートにサービスを登録
        m_port.registerProvider("Robo0", "MyRobot",
                               m_robot);

        // ポートをコンポーネントに登録
        registerPort(m_port);
    }
}
```

```
class MyRobotUser
{
private:
    // マニピュレータサービスのポートを宣言
    RTC::CorbaPort m_port;
    // サービスコンシューマのインスタンスを宣言
    RTC::CorbaConsumer<MyRobot> m_robot;
public:
    any_functions()
    {
        // サービスの利用例
        // ゲインをセット
        m_robot->setPosCtrlGain(0, 1.0);
        // ゲインを表示
        std::cout << m_robot->get_pos_ctrl_gain(i) <<
            std::endl;
    }
}
```



より詳細な実装については、
サンプルのSimpleServiceを参照

Configuration

- 複数のパラメータを持つ可能性のあるコンポーネントで、動的にパラメータ変更を行うことができる機能

Rtc-template 引数 (--config=[名前]:[型]:[デフォルト値]) で指定例

```
--config=int_param0:int:0 --config=double_param0:double:1.1
```

※これらは”default”という名前のConfigurationSetとしてソースに埋め込まれる

- rtc-link上でConfiguration Setを変更可能
- 詳しくはConfigSampleを参照

Configurationの実装例

ヘッダ
変数宣言

```
int m_int_param0;  
double m_double_param0;
```

istream operator>>が
定義されている型であ
ればどんな型でも可能

実装ファイル
先頭部分: spec定義にて

```
static const char* configsample_spec[] = {  
    :中略  
    "conf.default.int_param0", "0",  
    "conf.default.double_param0", "1.1",  
    :中略};
```

onInitialize()にて

```
bindParameter("int_param0", m_int_param0, "0");  
bindParameter("double_param0", m_double_param0, "1.1");
```

Configuration

rtc.confにて

: 略

category.component.config_file: comp.conf

: 略

comp.confにて

conf.**mode0**.int_param0: 2

conf.**mode0**.double_param0: 3.14

conf.**mode1**.int_param0: 3

conf.**mode1**.double_param0: 6.28

conf.**mode2**.int_param0: 4

conf.**mode2**.double_param0: 12.56

rtc-templateで自動的に生成され埋め込まれる

| | | | | | | | | | |
|---------|----|--|--|--|--|--|--|--|--|
| default | 名前 | | | | | | | | |
| | 値 | | | | | | | | |

| | | | | | | | | | |
|-------|----|--|--|--|--|--|--|--|--|
| mode0 | 名前 | | | | | | | | |
| | 値 | | | | | | | | |

| | | | | | | | | | |
|-------|----|--|--|--|--|--|--|--|--|
| mode1 | 名前 | | | | | | | | |
| | 値 | | | | | | | | |

| | | | | | | | | | |
|-------|----|--|--|--|--|--|--|--|--|
| mode2 | 名前 | | | | | | | | |
| | 値 | | | | | | | | |

コンポーネントのconfigファイルで追加することもできる。
(defaultセット同様ソースに埋め込むことも可能)

提言

- 自前主義はやめよう！！
 - 書きたてのコードより、いろいろな人に何万回も実行されたコードのほうが動くコードである！！
 - 自分にとって本質的でない部分は任せて、本当にやりたい部分・やるべき部分のコードを書こう！！
 - 誰かがリリースしたプログラムは一度は動いたことがあるプログラムである！！
 - 人のコードを読むのが面倒だからと捨ててしまうのはもったいない！！
- オープンソースにコミットしよう！！
 - 臆せずMLやフォーラムで質問しよう！！
 - どんなに初歩的な質問でも他の人にとっては価値ある情報である。
 - 要望を積極的にあげよう！！
 - できればデバッグしてパッチを送ろう！

まとめ

- RTミドルウェアの概要
 - 基本概念
 - モジュール化
 - 標準化
 - RTMコミュニティ