RTミドルウエア・サマーキャンプ2012

RTコンポーネントの開発手順について

日時: 2012年7月31日（火）
場所: 産総研中央第2 ネットワーク会議室

(独)産業技術総合研究所 知能システム研究部門
安藤慶昭

概要

- RTミドルウエア概要
- RTコンポーネントの作り方
- OpenRTM-aist-1.1.0の新機能
- 今後の展望

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ミドルウエア

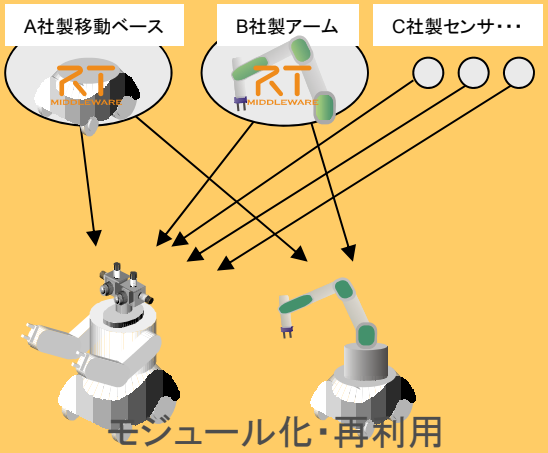
OpenRTM-aist

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

RTミドルウエアの目的

モジュール化による問題解決

コストの問題

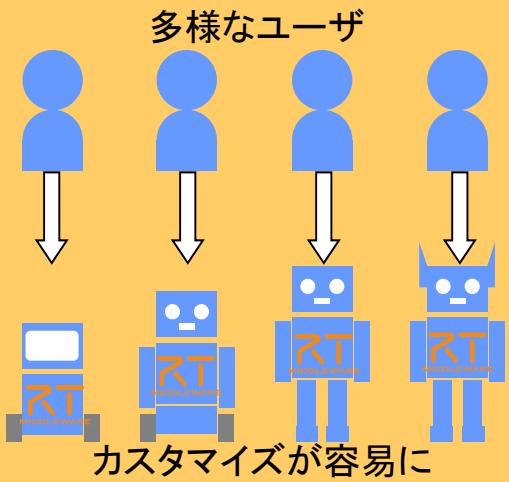


ロボットの低コスト化

技術の問題

最新技術を利用可能

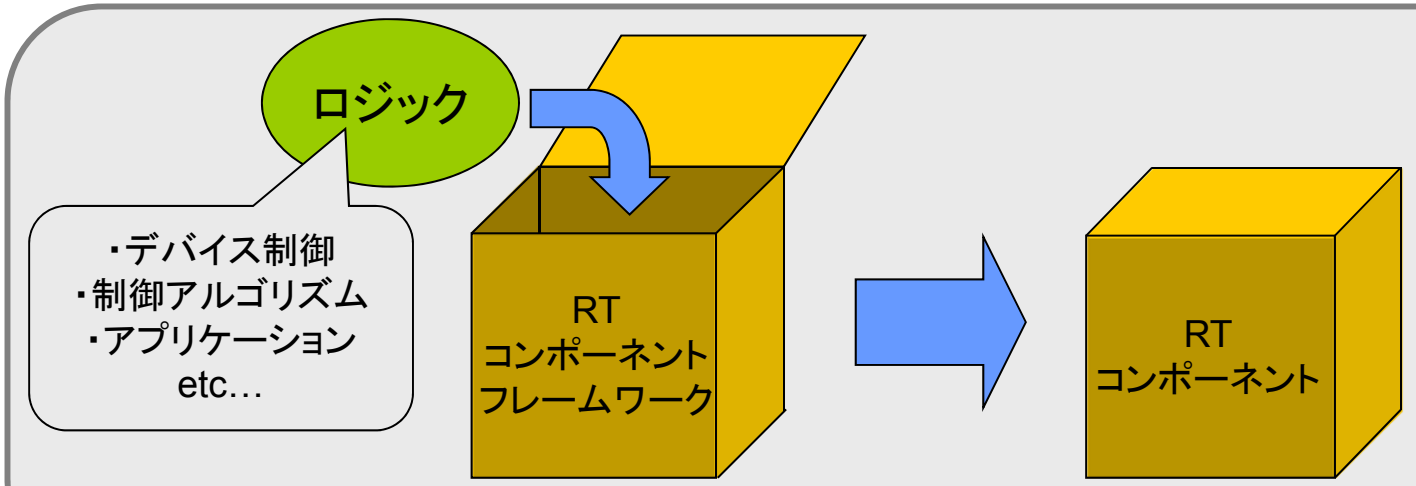
ニーズの問題



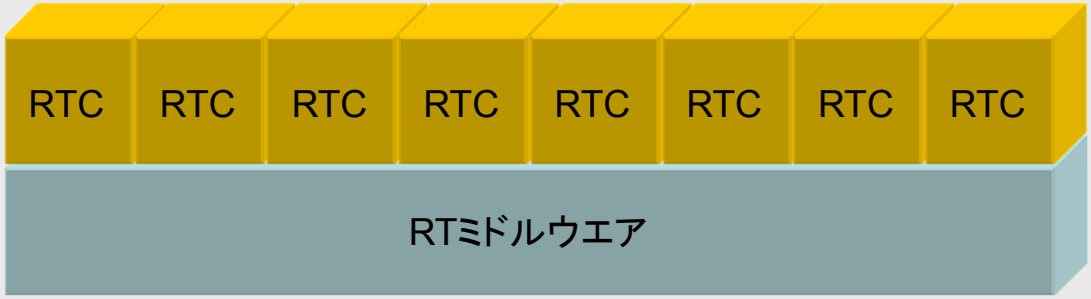
多様なニーズに対応

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

RTミドルウエアとRTコンポーネント



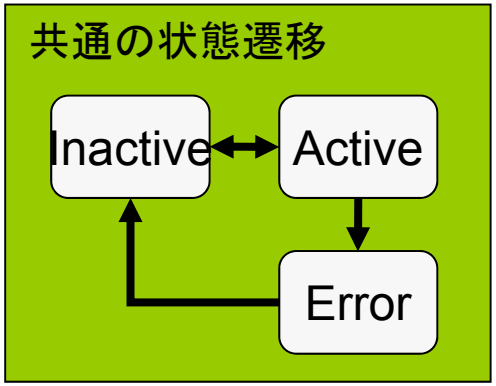
ロジックを箱（フレームワーク）に入れたもの＝RTコンポーネント（RTC）

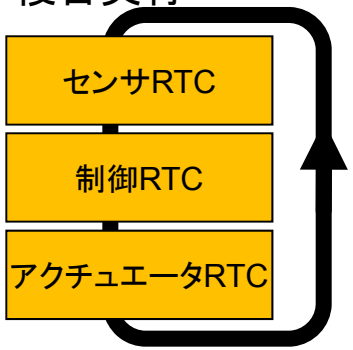


RTCの実行環境（OSのようなもの）＝RTミドルウエア（RTM）
※RTCはネットワーク上に分散可能

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

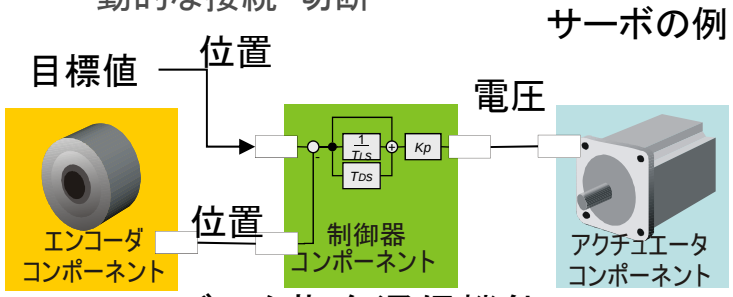




ライフサイクルの管理・コアロジックの実行

データポート

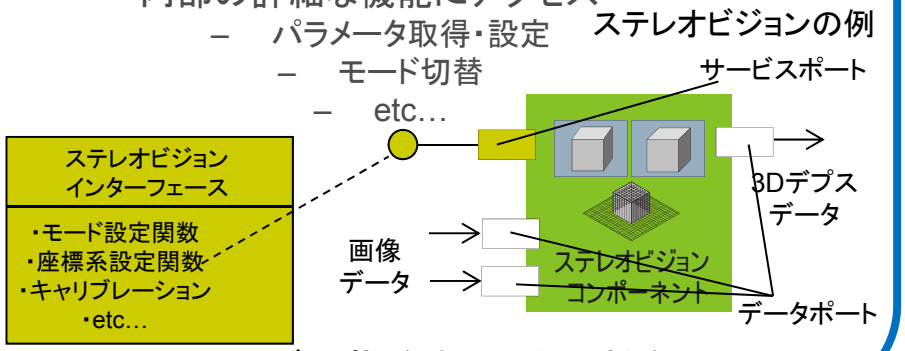
- データ指向ポート
- 連続的なデータの送受信
- 動的な接続・切断



データ指向通信機能

サービスポート

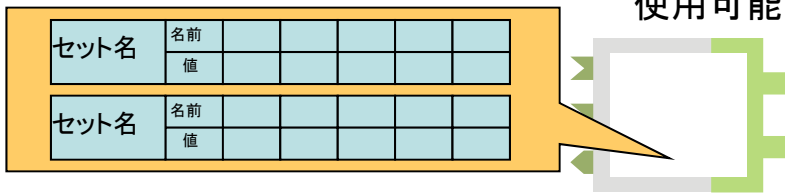
- 定義可能なインターフェースを持つ
- 内部の詳細な機能にアクセス
  - パラメータ取得・設定
  - モード切替
  - etc…



サービス指向相互作用機能

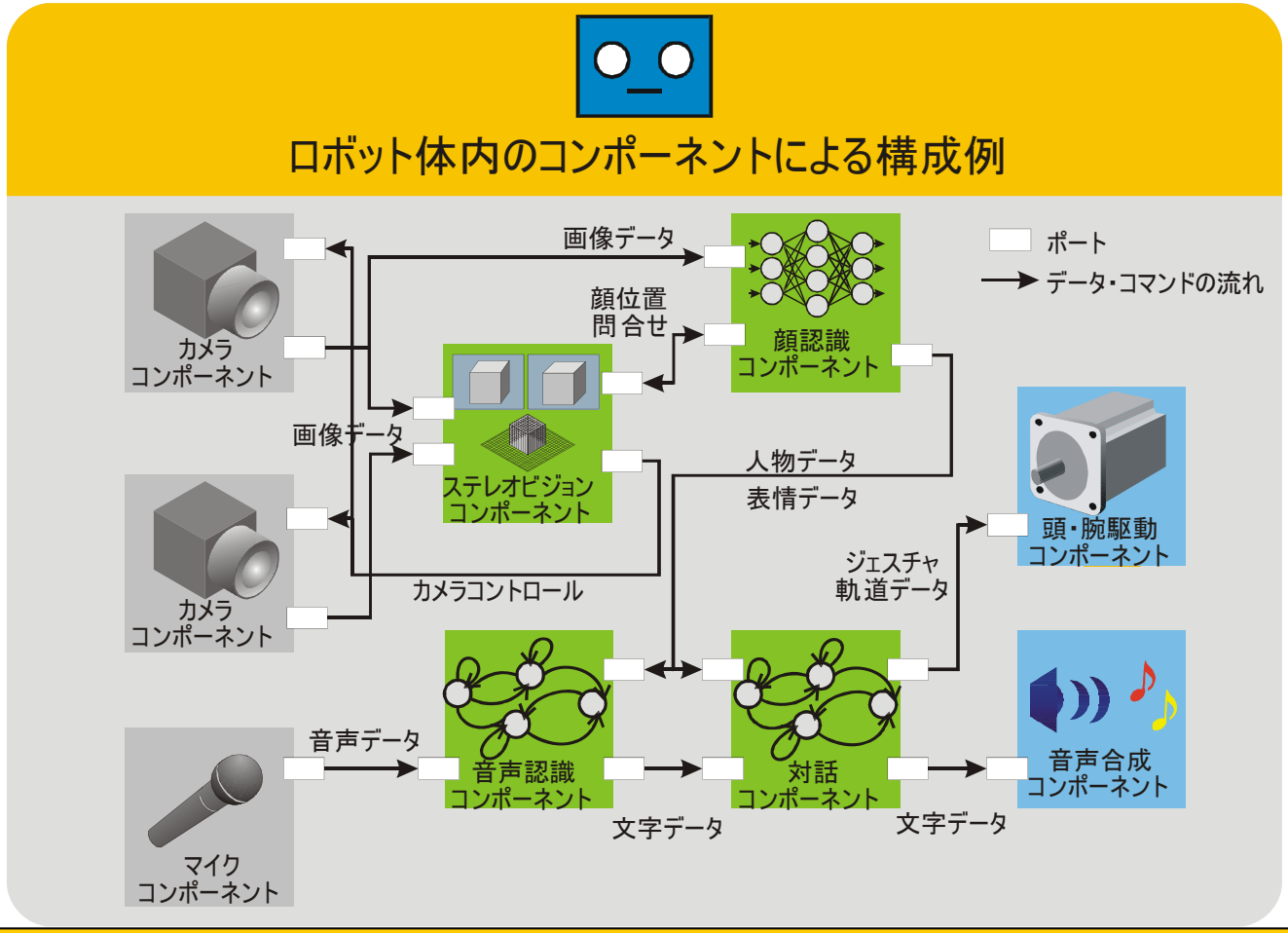
コンフィギュレーション

- 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

RTCの分割と連携



（モジュール）情報の隠蔽と公開のルールが重要

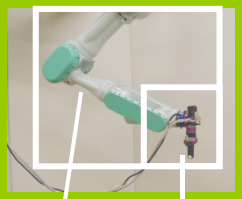
RTミドルウエアによる分散システム

RTMにより、ネットワーク上に分散するRTCをOS・言語の壁を越えて接続することができる。

ロボットA

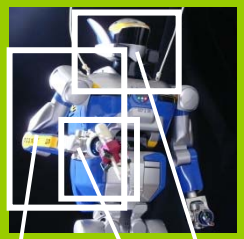
RTC | RTC
RTM
Solaris

ロボットB



RTC | RTC
RTM
FreeBSD

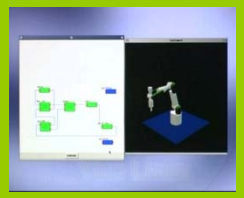
ロボットC



RTC | RTC | RTC
RTM
ARTLinux

ネットワーク

Linux
RTM
RTC | RTC



アプリケーション

Windows
RTM
RTC | RTC

操作デバイス

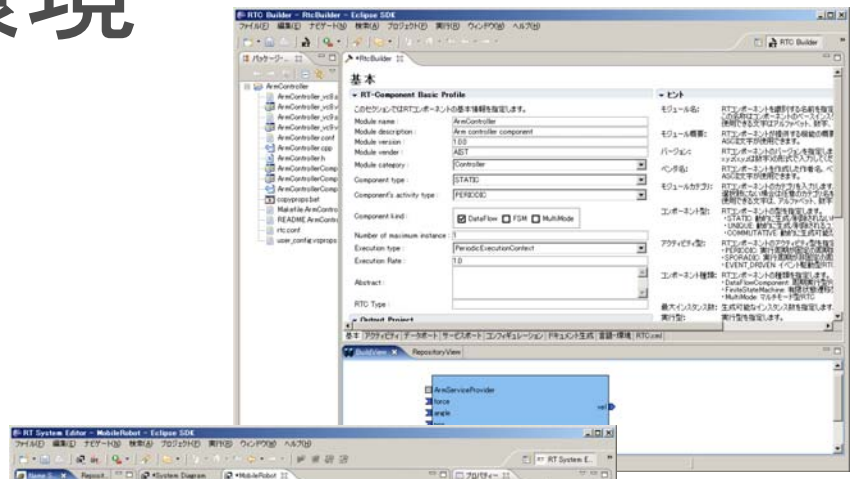
TRON
RTM
RTC | RTC

センサ

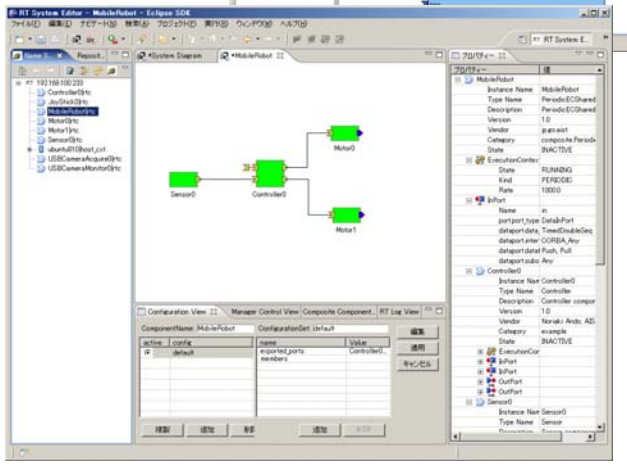
RTC同士の接続は、プログラム実行中に動的に行うことが出来る。

開発環境

- RTCBuilder (GUI版)
- rtc-template (CUI)
  - RTコンポーネントのコードジェネレータ
  - GUI画面で必要事項を入力
  - C++, Python, Java, C#等のコードを自動生成
- RTSystemEditor
  - ネットワーク上のすべてのコンポーネントの操作が可能
  - コンポーネントのON/OFF、パラメータの変更、状態監視
  - コンポーネント間の接続



RTCBuilder



RTSystemEditor

RTC・RTM統合開発環境の整備

RTC設計・実装・デバッグ、RTMによるインテグレーション・デバッグまでを一貫して行うことができる統合開発環境をEclipse上に構築

OpenRTM-aist

- コンポーネントフレームワーク + ミドルウエアライブラリ
- コンポーネントインターフェース:
  - OMG Robotic Technology Component Specification ver1.0 準拠
- OS
  - 公式：FreeBSD, Linux (Fedora, Debian, Ubuntu, Vine, Scientific), Windows
  - 非公式：Mac OS X, uITRON, T-Kernel, VxWorks
- 言語:
  - C++ (1.1.0), Python (1.0.0), Java (1.0.0)
  - .NET (implemented by SEC)
- CPU アーキテクチャ (動作実績):
  - i386, ARM9, PPC, SH4
  - PIC, dsPIC, H8 (RTC-Lite)
- ツール (Eclipse プラグイン)
  - テンプレートソースジェネレータ: rtc-template、RTCBuilder
  - システムインテグレーションツール: RTSystemEditor
  - その他
    - Pattern weaver for RT-Middleware (株式会社テクノロジックアートより発売中)

OpenRTMの利点

- 共通コンポーネントフレームワークを提供
  - OMG標準
  - コールバックベースの枠組み、共通状態マシン、複合化に対応
  - 大部分のコード生成を自動化
- 多言語対応
  - C++, Java, Python, .NET (by SEC)
- 多様なOSへのネイティブ対応
  - FreeBSD, Linux, Mac OS X, Windows
  - 試験的:TOPPERS, T-Kernel, VxWorks
- ツールの提供
  - Eclipseベースのツール群 (RTCB, RTSE)
  - コマンドラインツール群 (rtchell)
- デュアルライセンス (LGPL・EPLと個別ライセンス)
  - RTCにはラインセンスが及ばない（RTCのバイナリ供給が可能に）
  - 商用化、事業化、組込み用途には個別ライセンスで対応

OMG RTC ファミリ

Name	Vendor	Feature
OpenRTM-aist	AIST	C++, Python, Java
OpenRTM.NET	SEC	.NET(C#,VB,C++/CLI, F#, etc..)
miniRTC, microRTC	SEC	CAN・ZigBee等を利用した組込用RTC実装
Dependable RTM	SEC/AIST	機能安全認証 (IEC61508) capableなRTM実装
RTC CANOpen	SIT, CiA	CANOpenのためのCiA (Can in automation) におけるRTC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

同一標準仕様に基づく多様な実装により
- 実装(製品)の継続性を保証
- 実装間での相互利用がより容易に

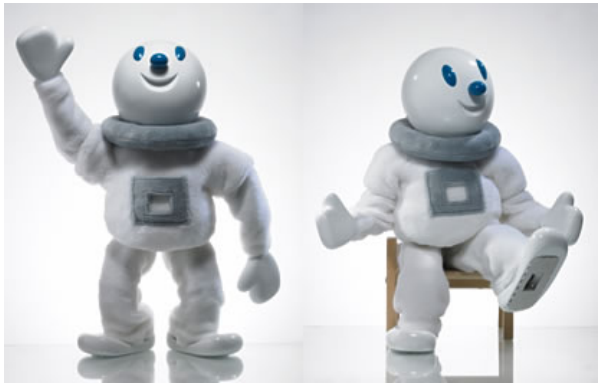
実用化・事業化

HRP-4: Kawada/AIST

DAQ-Middleware: KEK/J-PARC

KEK: High Energy Accelerator Research Organization

J-PARC: Japan Proton Accelerator Research Complex



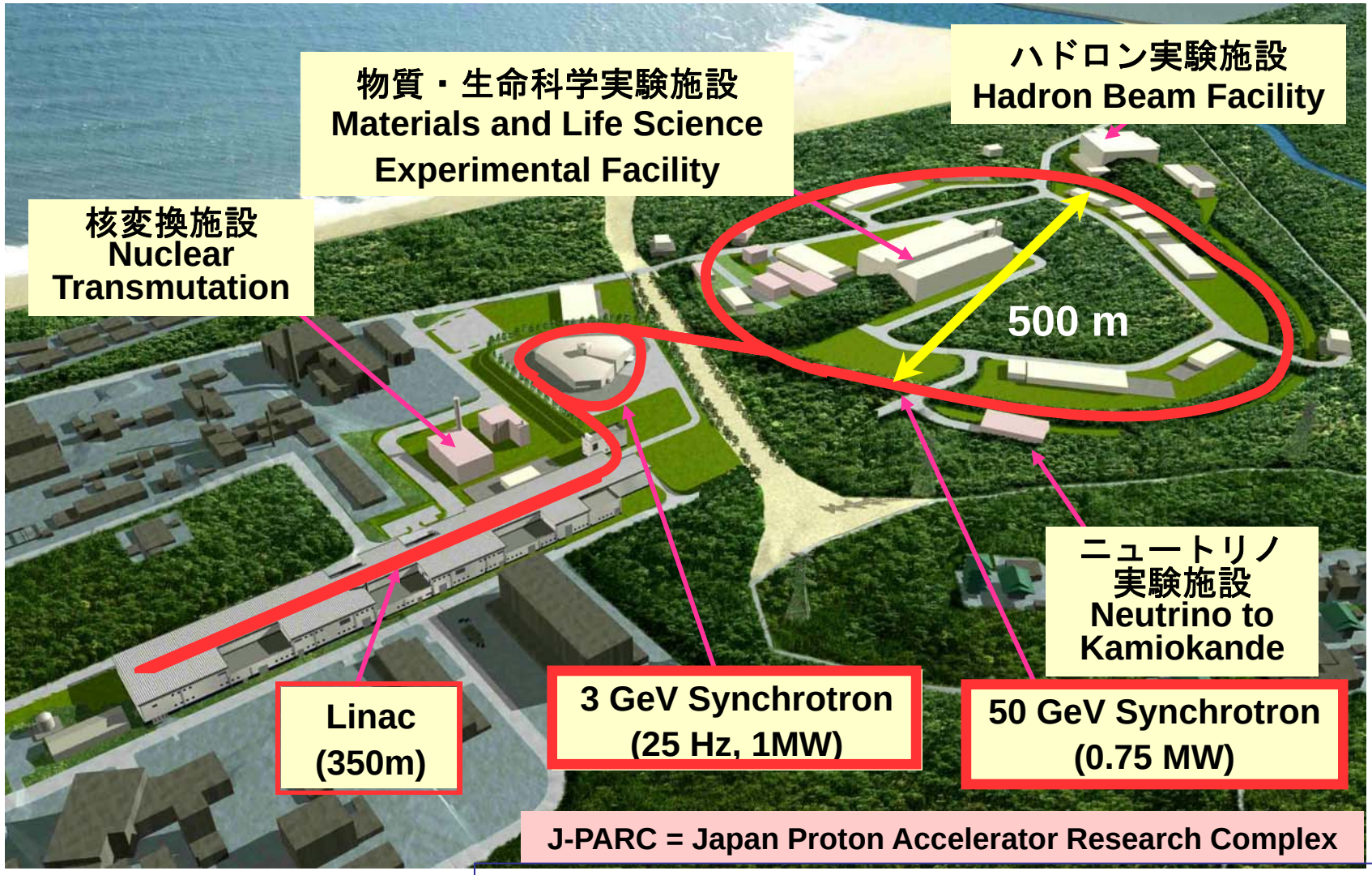
TAIZOU: General Robotics Inc.

HIRO: Kawada/GRX

HRP-4C: Kawada/AIST

Japan Proton Accelerator Research Complex

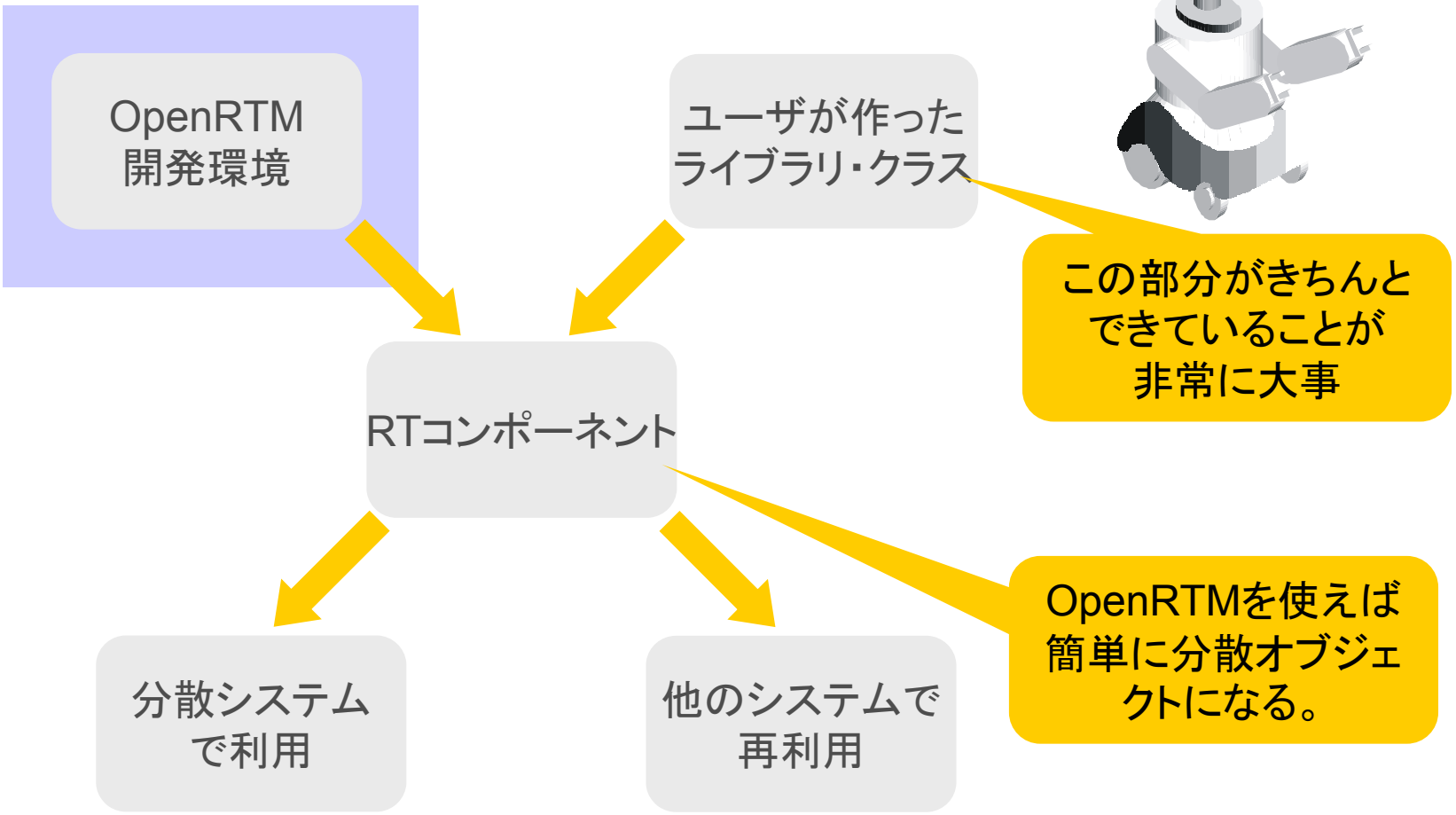
（J-PARC,大強度陽子加速器施設）



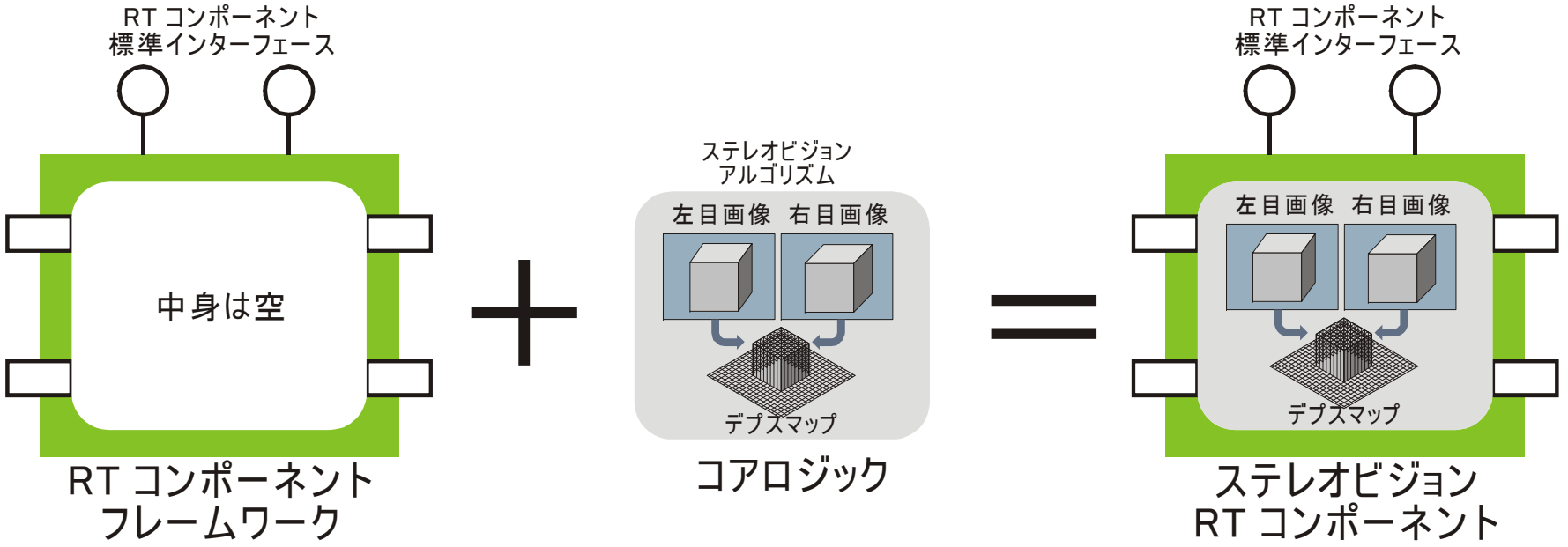
Joint Project between KEK and JAEA (former JAERI)

RTコンポーネントプログラミング

OpenRTMを使った開発の流れ

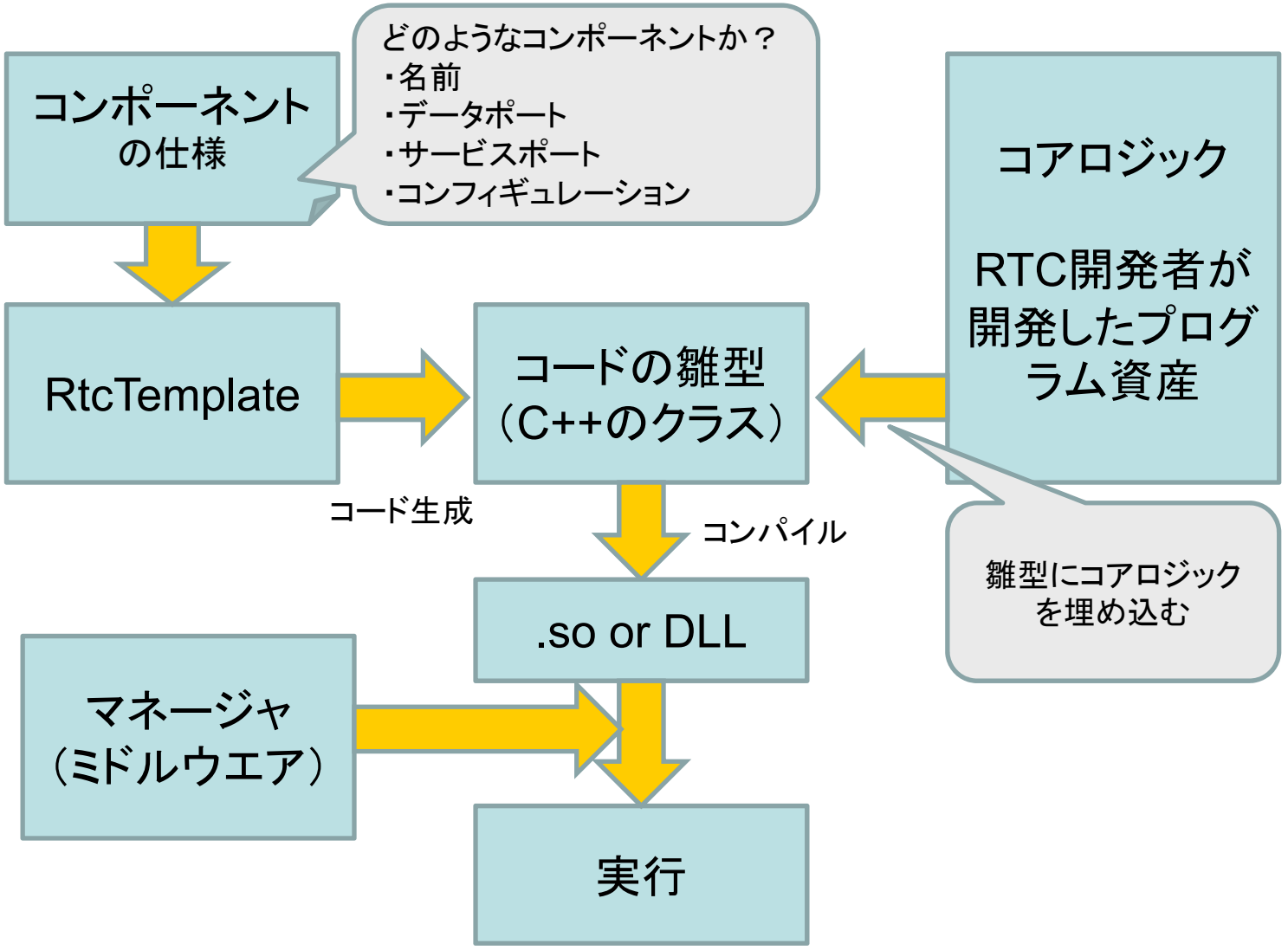


フレームワークとコアロジック



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

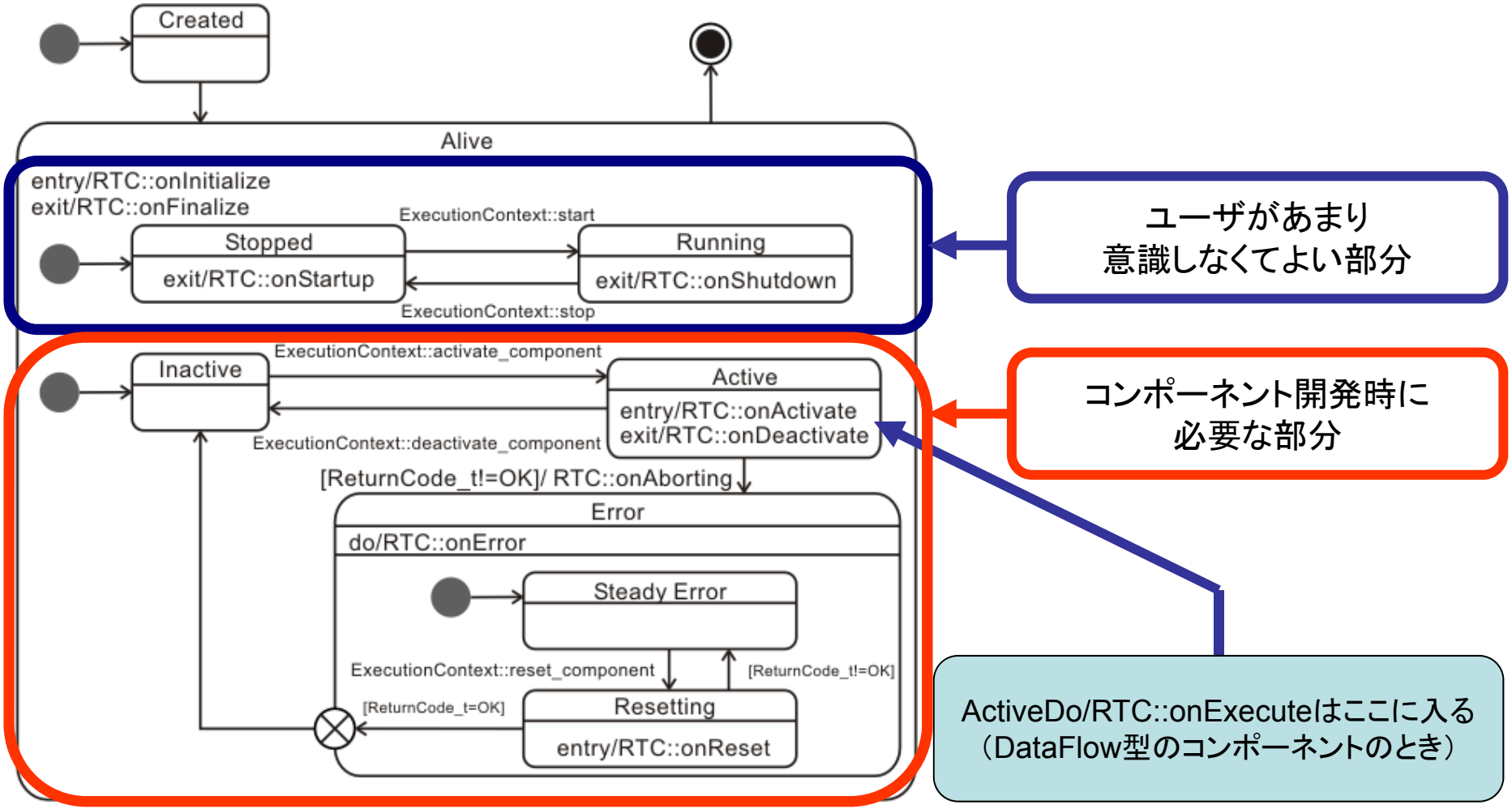


コード例

- 生成されたクラスのメンバー関数に必要な処理を記述
- 主要な関数
  - onExecute (周期実行)
- 処理
  - InPortから読む
  - OutPortへ書く
  - サービスを呼ぶ
  - コンフィギュレーションを読む

```
class MyComponent
 : public DataflowComponentBase
{
public:
  // 初期化時に実行したい処理
  virtual ReturnCode_t onInitialize()
  {
    if (mylogic.init())
      return RTC::RTC_OK;
    return RTC::RTC_ERROR;
  }

  // 周期的に実行したい処理
  virtual ReturnCode_t onExecute(RTC::UniqueId ec_id)
  {
    if (mylogic.do_someting())
      return RTC::RTC_OK;
    RTC::RTC_ERROR;
  }

private:
  MyLogic mylogic;
  // ポート等の宣言
  //   :
};
```

コンポーネント内の状態遷移



コールバック関数

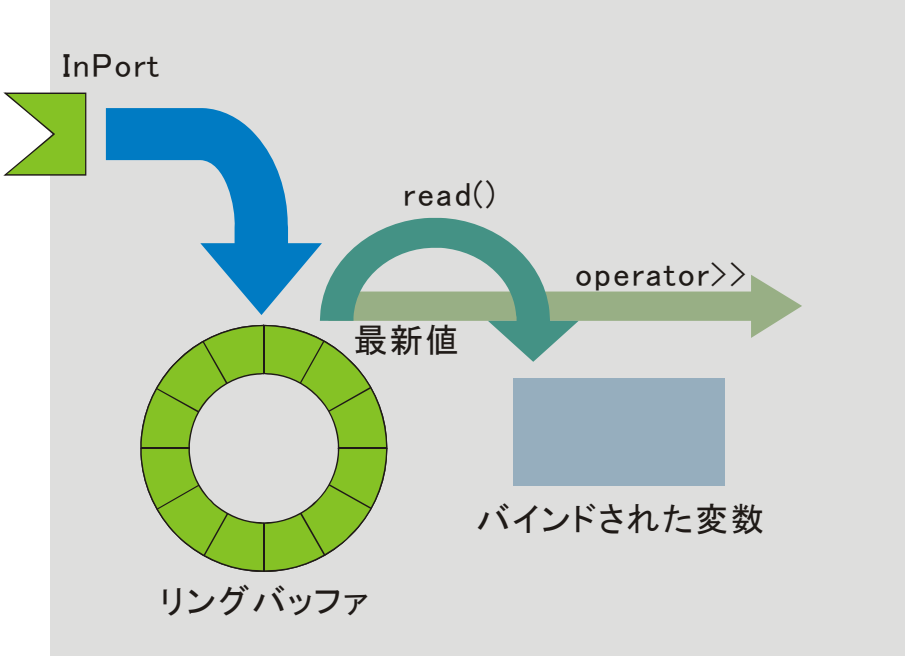
RTCの作成＝コールバック関数に処理を埋め込む

コールバック関数	処理
onInitialize	初期化処理
onActivated	アクティブ化されるとき1度だけ呼ばれる
onExecute	アクティブ状態時に周期的に呼ばれる
onDeactivated	非アクティブ化されるとき1度だけ呼ばれる
onAborting	ERROR状態に入る前に1度だけ呼ばれる
onReset	resetされる時に1度だけ呼ばれる
onError	ERROR状態のときに周期的に呼ばれる
onFinalize	終了時に1度だけ呼ばれる
onStateUpdate	onExecuteの後毎回呼ばれる
onRateChanged	ExecutionContextのrateが変更されたとき1度だけ呼ばれる
onStartup	ExecutionContextが実行を開始するとき1度だけ呼ばれる
onShutdown	ExecutionContextが実行を停止するとき1度だけ呼ばれる

とりあえずは
この5つの関数
を押さえて
おけばOK

InPort

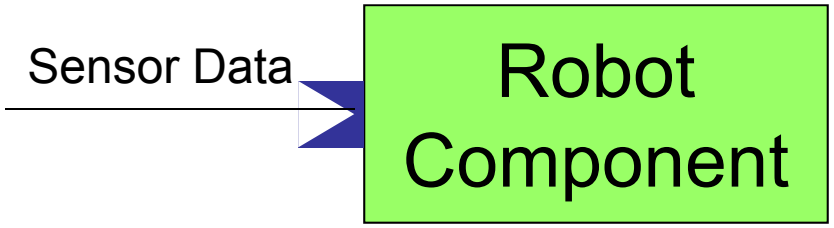
- InPortのテンプレート第2引数：バッファ
  - ユーザ定義のバッファが利用可能
- InPortのメソッド
  - read(): InPort バッファからバインドされた変数へ最新値を読み込む
  - >> : ある変数へ最新値を読み込む



基本的にOutPortと対になる

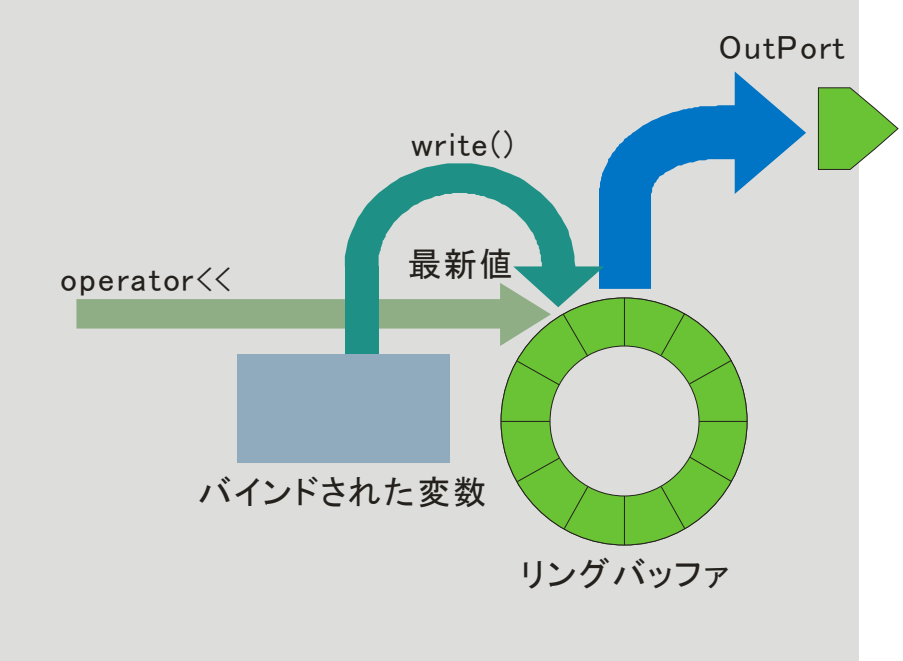
データポートの型を
同じにする必要あり

例



OutPort

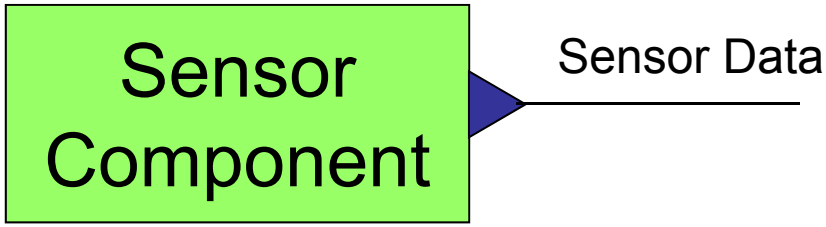
- OutPortのテンプレート第2引数：バッファ
  - ユーザ定義のバッファが利用可能
- OutPortのメソッド
  - write(): OutPort バッファへバインドされた変数の最新値として書き込む
  - >> : ある変数の内容を最新値としてリングバッファに書き込む



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;
 };
```

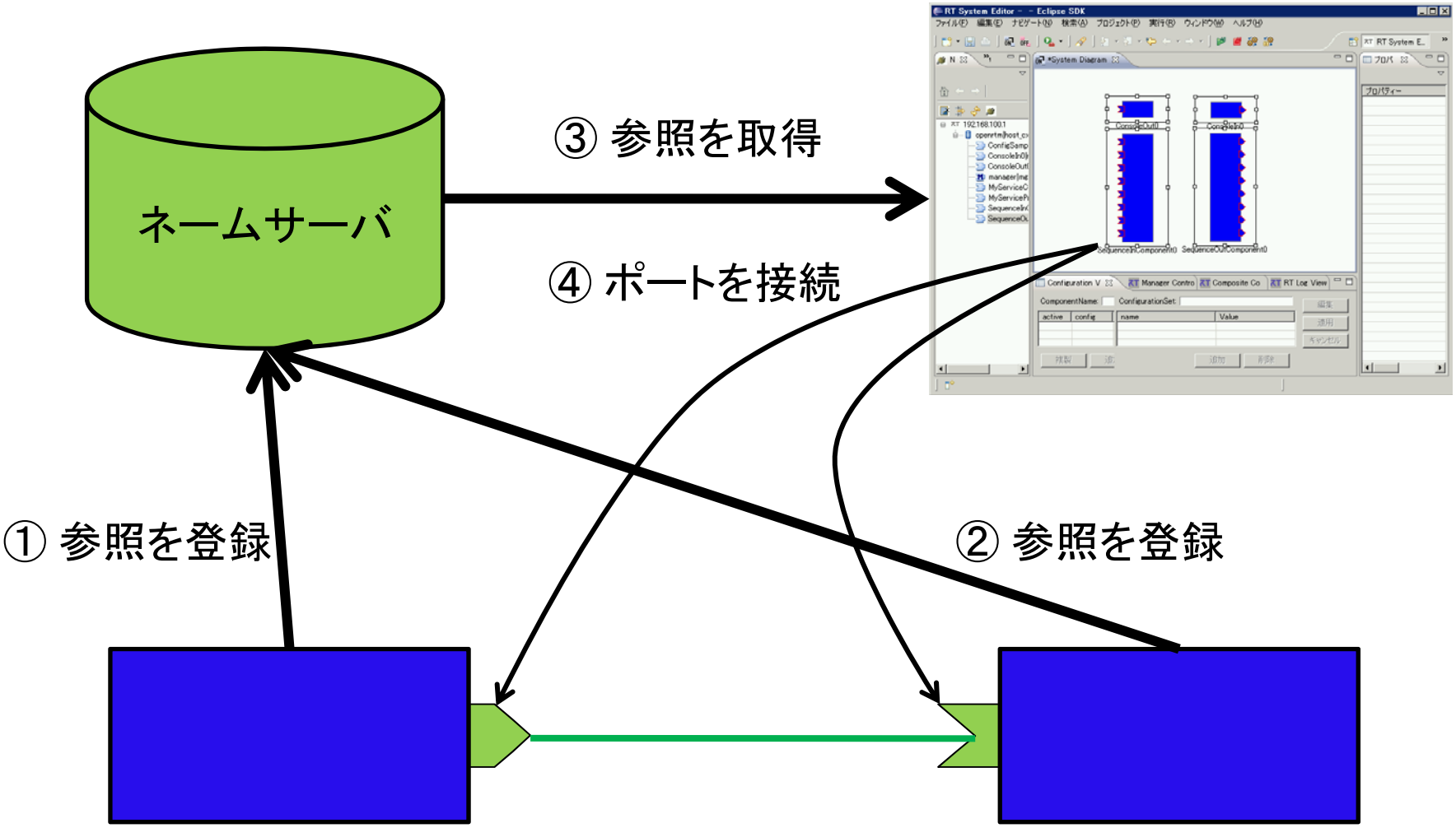
- 基本型
  - tm：時刻
  - data: データそのもの

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

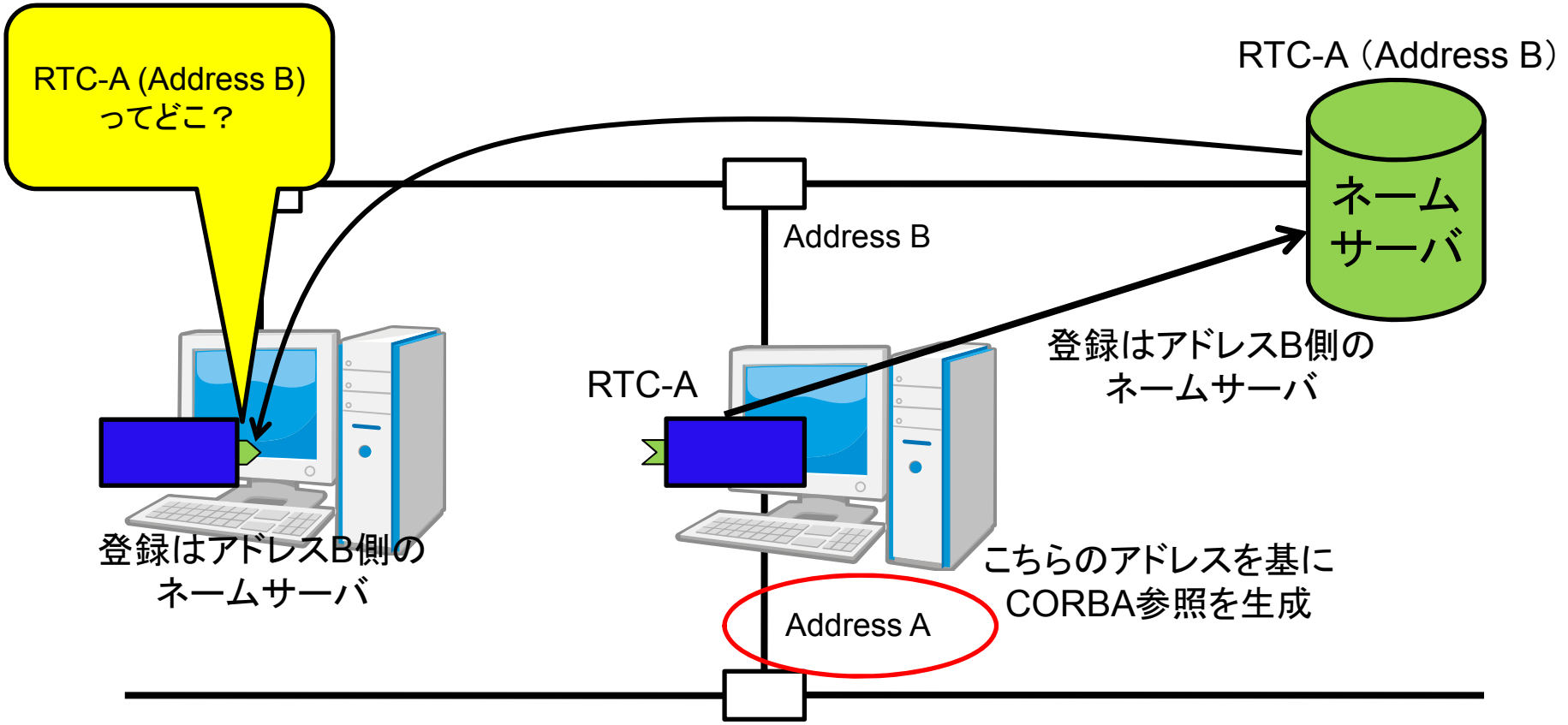
- シーケンス型
  - data[i]: 添え字によるアクセス
  - data.length(i): 長さiを確保
  - data.length(): 長さを取得

- 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

動作シーケンス



ネットワークインターフェースが2つある場合の注意



Rtc.confについて

RT Component起動時の登録先NamingServiceや、登録情報などについて記述するファイル

記述例：

corba.nameservers: localhost:9876

naming.formats: SimpleComponent/%n.rtc

（詳細な記述方法は etc/rtc.conf.sample を参照）

以下のようにすると、コンポーネント起動時に読み込まれる

./ConsoleInComp –f rtc.conf

ネーミングサービス設定

corba.nameservers	host_name:port_numberで指定、デフォルトポートは2809(omniORBのデフォルト)、複数指定可能
naming.formats	%h.host_cxt/%n.rtc →host.host_cxt/MyComp.rtc 複数指定可能、0.2.0互換にしたければ、 %h.host_cxt/%M.mgr_cxt/%c.cat_cxt/%m.mod_cxt/%n.rtc
naming.update.enable	“YES” or “NO”: ネーミングサービスへの登録の自動アップデート。コンポーネント起動後にネームサービスが起動したときに、再度名前を登録する。
naming.update.interval	アップデートの周期[s]。デフォルトは10秒。
timer.enable	“YES” or “NO”: マネージャタイマ有効・無効。 naming.updateを使用するには有効でなければならない
timer.tick	タイマの分解能[s]。デフォルトは100ms。

必須の項目　　必須でないOption設定

ログ設定

logger.enable	“YES” or “NO”: ログ出力を有効・無効
logger.file_name	ログファイル名。 %h：ホスト名、%M:マネージャ名,%p：プロセスID 使用可
logger.date_format	日付フォーマット。strftime(3)の表記法に準拠。 デフォルト：%b %d %H:%M:%S → Apr 24 01:02:04
logger.log_level	ログレベル： SILENT, ERROR, WARN, NORMAL, INFO, DEBUG, TRACE, VERBOSE, PARANOID SILENT：何も出力しない PARANOID：全て出力する ※以前はRTC内で使えましたが、現在はまだ使えません。

必須の項目　　必須でないOption設定

その他

corba.endpoints	IP_Addr:Port で指定：NICが複数あるとき、ORBをどちらでlistenさせるかを指定。Portを指定しない場合でも":"が必要。 例 "corba.endpoints: 192.168.0.12:" NICが2つある場合必ず指定。 (指定しなくても偶然正常に動作することもあるが念のため。)
corba.args	CORBAに対する引数。詳細はomniORBのマニュアル参照。
[カテゴリ名]. [コンポーネント名]. config_file または [カテゴリ名]. [インスタンス名]. config_file	コンポーネントの設定ファイル •カテゴリ名：manipulator, •コンポーネント名：myarm, •インスタンス名myarm0,1,2,… の場合 manipulator.myarm.config_file: arm.conf manipulator.myarm0.config.file: arm0.conf のように指定可能

使いたいNICに割り当てられているIPアドレス

必須の項目　　必須でないOption設定

OpenRTM-aist-1.1.0の新機能

OpenRTM-aist-1.1.0の新機能(1)

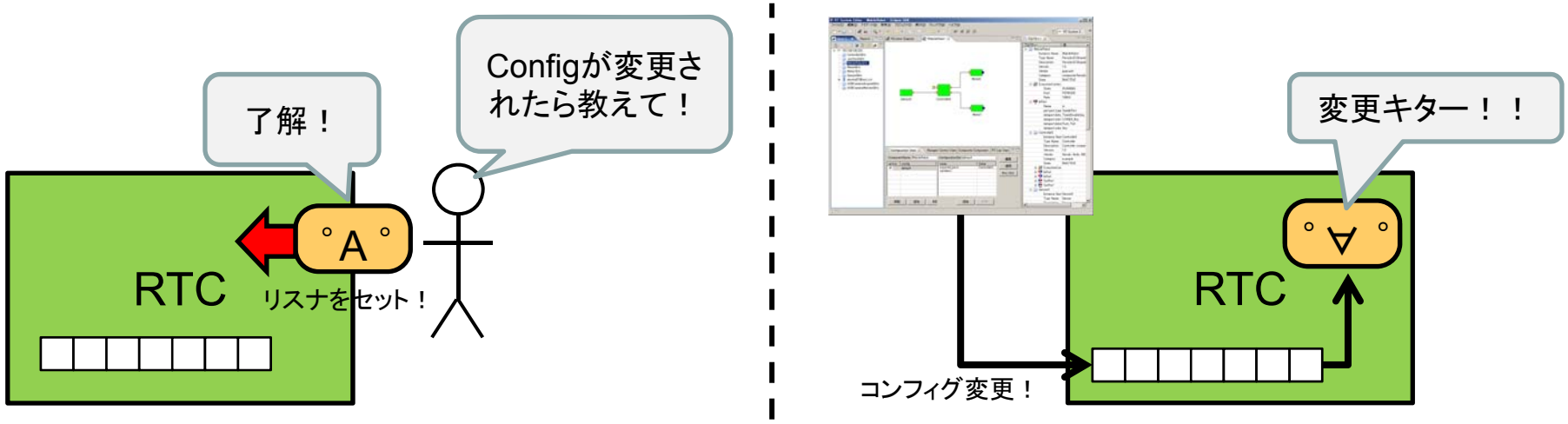
- コールバック機能強化
  - ComponentActionListener
  - PortConnectionListener
  - ManagerActionListener
  - ConfigurationListener
  - ConnectorListener
- SDOサービスフレームワークの導入
  - Observerの導入：RTSEの高速化
- 内部APIの追加
- 雑多なバグフィックス

OpenRTM-aist-1.1.0の新機能(2)

- ライセンスの変更：EPL→LGPL
- 64bit対応（Linux、Windows）
- OS・CORBA対応
  - Mac OS X, RtORB (Linux, Cygwin, Mac)対応
- VC2010対応
- Cmake対応
- RT preemptive real-time Linux kernel 用ECの正式サポート
- Deb,rpmパッケージ作成機能提供

コールバック機能

- コンポーネント内で発生する様々なイベントに対してあるアクションを行う機能
  - 例：コンフィグパラメータが変更されたら、画面の表示を更新する
  - 例：ポートが接続されたら、実際に計算を行ってデータを出力、等
- 開発者が予めセットしておいたリスナオブジェクトの特定の関数がイベント発生時に呼ばれる。



イベントタイプ（1）

タイプ	概要	ヘッダ
コンポーネント本体に関連するイベント		
PreComponentActionListenerType	ComponentAction (onInitialize()等)の実行直前に発生するイベント	ComponentActionListener.h
PostComponentActionListenerType	ComponentAction (onInitialize()等)の実行直後に発生するイベント	ComponentActionListener.h
PortActionListenerType	ポートの追加、削除イベント	ComponentActionListener.h
ExecutionContextActionListenerType	実行コンテキストのアタッチ、でタッチなどのイベント	ComponentActionListener.h
コンフィギュレーションパラメータに関連するイベント		
ConfigurationParamListenerType	コンフィグパラメータの更新操作時のイベント	ConfigurationListener.h
ConfigurationSetListenerType	コンフィグパラメータセットの操作時のイベント	ConfigurationListener.h
ConfigurationSetNameListenerType	コンフィグパラメータセットの操作時にイベント	ConfigurationListener.h

イベントごとに、イベントタイプやリスナクラスの型が異なる

イベントタイプ（2）

タイプ	概要	ヘッダ
ポート内部の振る舞いに関連するイベント		
InPort read系 (旧式)	InPortに対してreadを行う際に発生するイベント	PortCallback.h
OutPort write系 (旧式)	OutPortに対してwriteを行うに発生するイベント	PortCallback.h
PortConnectListenerType	ポートの接続時の各種処理に関係するイベント (notify_(dis)connect(), unsubscribeinterfaces()等)	PortConnectListener.h
PortConnectRetListenerType	ポートの接続時の各種処理に関係するイベント (connect_next(), subscribeinterfaces(), 接続切断完了通知)	PortConnectListener.h
ConnectorDataListenerType	データポートのコネクタ内のイベント (bufferフル, send/received などの完了やエラー通知)	ConnectorListener.h
ConnectorListenerType	データポートのコネクタ内のイベント (buffer空, 接続・切断などの完了やエラー通知)	ConnectorListener.h

どのようなイベントがあるかは、クラスリファレンスか
イベントごとのヘッダファイル内のドキュメントを参照

使い方

1. 利用するイベントを決める
   - 例：Configurationの更新
2. イベントタイプを調べる
   - 例：ConfigurationParamListenerTypeのON_UPDATE_CONFIG_PARAMを利用
3. リスナの基底を継承してファンクタを実装
   - 例： ConfigurationParamListenerを継承してMyConfigUpdateParamを実装
4. onInitialize()などでリスナを登録
   - 例： addConfigurationParamListener()
5. イベント発生時にリスナがコールされる

コンポーネントのヘッダファイル

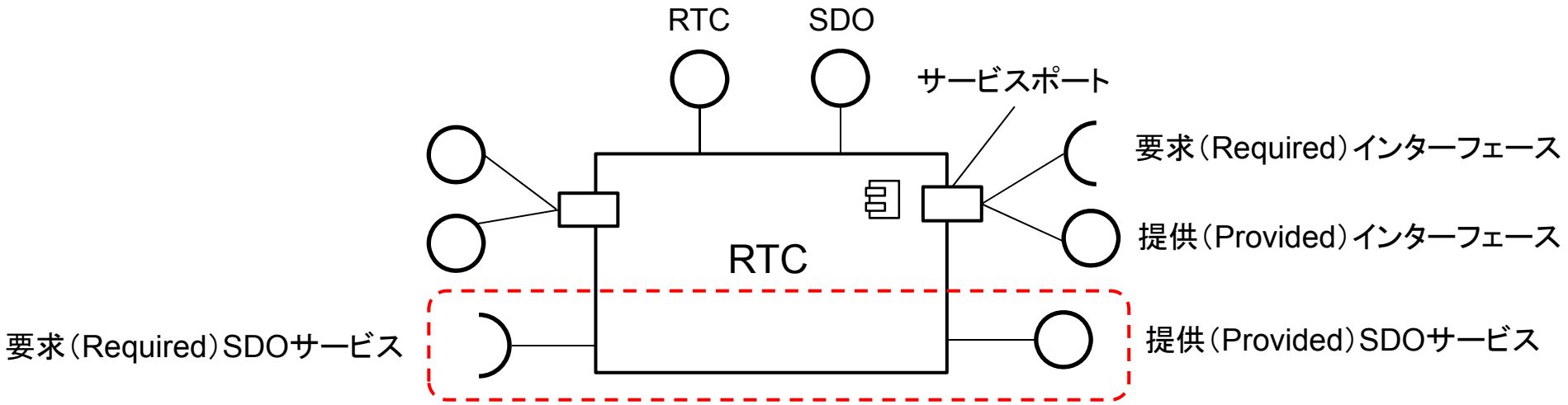
```
#include <rtm/ConfigurationListener.h>

class MyConfigUpdateParam
 : public ConfigurationParamListener
{
  virtual void operator()(const char* config_set_name,
                          const char* config_param_name)
  {
    std::cout << config_set_name << “の”
              << config_param_name
              << “が更新されました” << std::endl;
  }
}
```

コンポーネントの実装ファイル

```
RTC::ReturnCode_t ConsoleIn::onInitialize()
{
  : 中略
  addConfigurationParamListener(
              ON_UPDATE_CONFIG_PARAM,
              new MyConfigUpdateParam ());
}
```

SDOサービス

- SDO(Super distributed object)
  - RTCのベースとなっているOMG標準
- SDOサービス：ポートに属さないサービス
  - コンポーネント一般に関係するサービスを提供、要求する
    - 例：ロギング、デバッグ、パフォーマンス計測等
    - 特定のRTCの特定の機能（例：ナビゲーションアルゴリズム）に関係するサービスはサービスポートで提供・要求する



新たなAPI

EC(Execution Context:実行コンテキスト)に関する操作

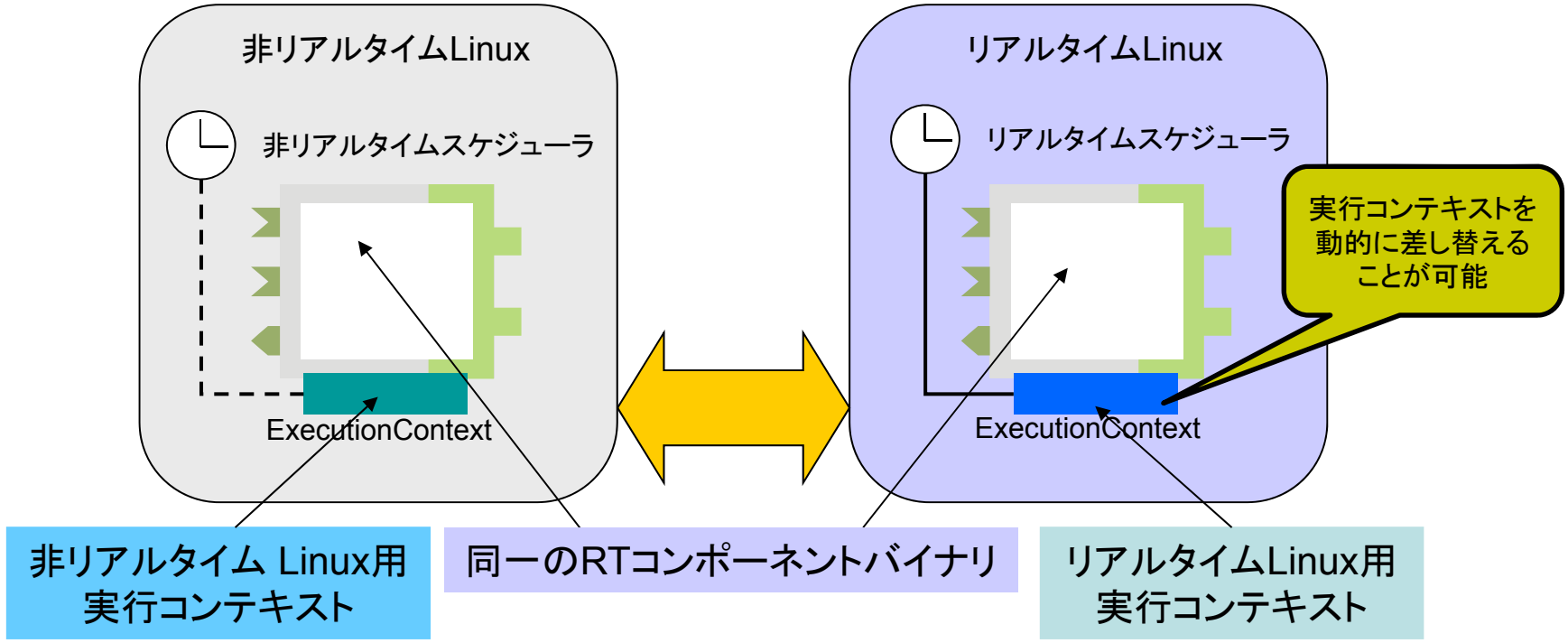
関数	意味
ExecutionContext_ptr getExecutionContext(RTC::UniqueId ec_id);	現在のECを取得
double getExecutionRate(RTC::UniqueId ec_id);	現在のECの実行周期を取得
ReturnCode_t setExecutionRate(RTC::UniqueId ec_id, double rate);	現在のECの実行周期をセット
bool isOwnExecutionContext(RTC::UniqueId ec_id);	現在のECが自身のECかどうか
ReturnCode_t deactivate(RTC::UniqueId ec_id);	現在のECでactive化する
ReturnCode_t activate(RTC::UniqueId ec_id);	現在のECで非active化する
ReturnCode_t reset(RTC::UniqueId ec_id);	現在のECでresetする

これらの関数は原則RTObject::onXXX() 関数内でのみ実行可能(詳細はリファレンスマニュアルを参照のこと)

リアルタイムECの提供

- 2つのリアルタイム実行コンテキスト（EC）
- ArtLinuxEC
  - ARTLinux用の実行コンテキスト
  - 1ms（orそれ以上）の精度でリアルタイム実行が可能
  - Ubuntu用のkernel debパッケージが利用可能なので、インストールし、rtc.confで利用するECをArtLinuxECに指定すれば利用可能
- PreemptEC
  - LinuxのPreemption Pathcedなkernelのリアルタイム機能を利用したEC
  - 1ms程度の精度でリアルタイム実行が可能
  - Ubuntu等では標準でrt-kernelとして提供されている

リアルタイム実行コンテキスト



非リアルタイムLinux環境で作られたRTコンポーネントを
再コンパイルせずにリアルタイムLinux上でリアルタイム実行可能

雑多な機能追加

- deb,rpmパッケージ作成機能
  - configure, cd packages, make でパッケージ作成
- データポートのPortProfile内データ型をIFRに変更
- RtORB（産総研CORBA）正式対応、Cygwin対応
- クラスリファレンスの拡充
- Version.txtの導入
- ロギング時のタイムスタンプをμsまで表示
  - See rtc.conf.sample

雑多なバグフィックス

- 標準ECのsleep時間計算方法の修正
  - 20ms以上の精度が必要な場合は PreemptiveECかArtLinuxECを利用してください
- Rtcdでゾンビプロセスが残る問題の修正
- マネージャ終了時の異常
- リングバッファのデッドロック
- 必要な箇所へのMutexの追加

Java版、Python版、ツール

Java、Python版

- 8月くらいまでに1.1.0-RELEASEを順次リリース
- 追加機能についてはC++と同じ
  - APIのシグニチャなどもC++と極力同じに
- リアルタイム機能は提供されない

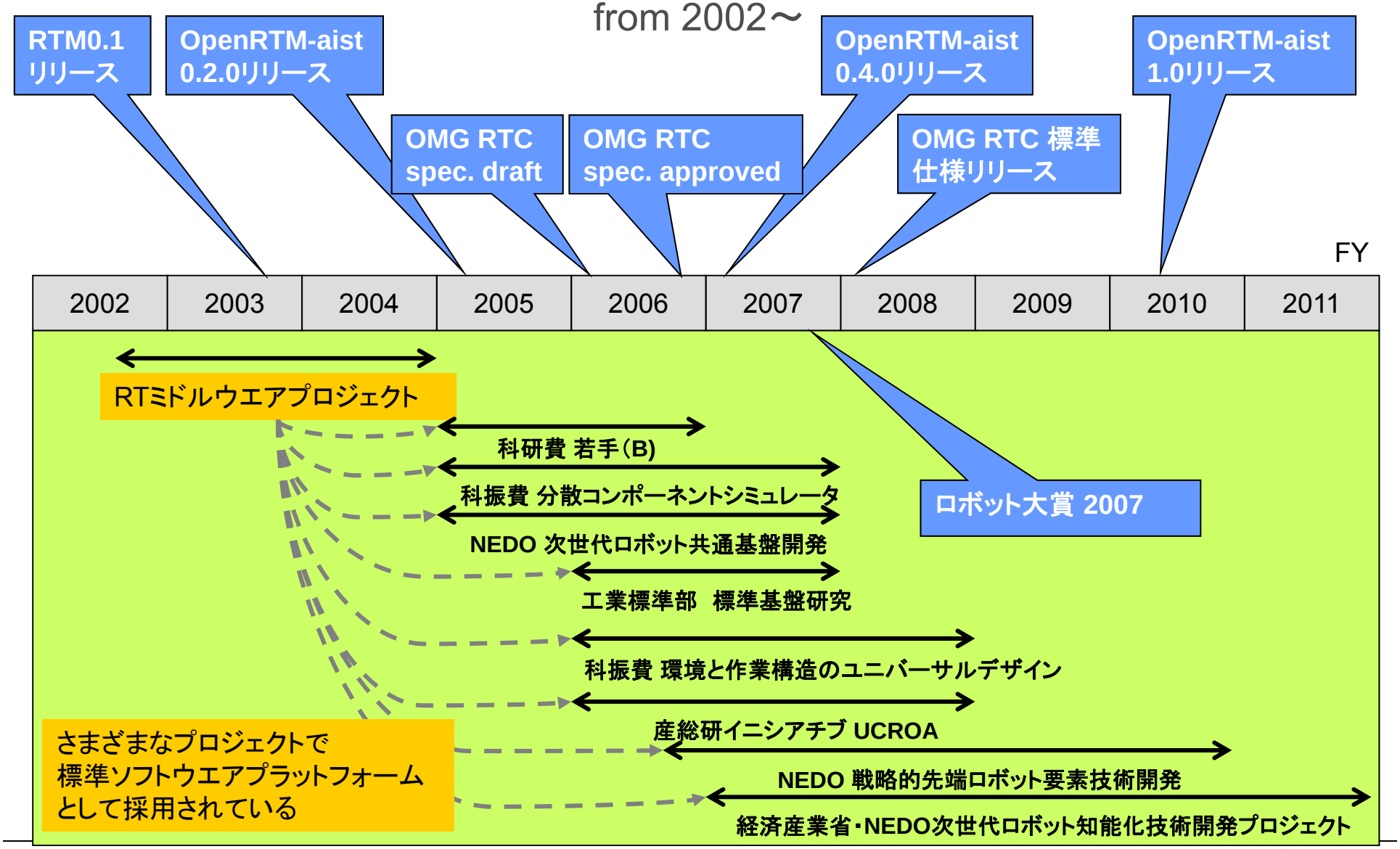
ツール

- Eclipse更新サイトを準備
- オンラインアップデートが可能に

今後の展望

RT-Middleware関連プロジェクト

from 2002～



NEDO知能化プロジェクト
（次世代ロボット知能化技術開発プロジェクト）

- 昨年度で終了
  - OpenRTM-aistの開発は今後も産総研・安藤が継続
- 成果を一般公開（ROBOSSA含む）
  - 多数のRTコンポーネントやツール
  - 原則として自由に利用可能に
  - オープンソース公開、バイナリ公開など様々
  - 継続的メンテナンスを持続する仕組みを検討中
    - 著作編集権の委譲、メンテナの一般からの募集等

ユーザの皆さま役割がより重要になります。
ご協力よろしくお願いします！！

RTミドルウエアの広がり

ダウンロード数

2012年2月現在

	2008年	2009年	2010年	2011年	2012年	合計
C++	4978	9136	12049	1851	253	28267
Python	728	1686	2387	566	55	5422
Java	643	1130	685	384	46	2888
Tool	3993	6306	3491	967	39	14796
All	10342	18258	18612	3768	393	51373

ユーザ数

タイプ	登録数
Webページユーザ	365 人
Webページアクセス	約 300 visit/day 約 1000 view/day
メーリングリスト	447 人
講習会	のべ 572 人
利用組織（Google Map）	46 組織

プロジェクト登録数

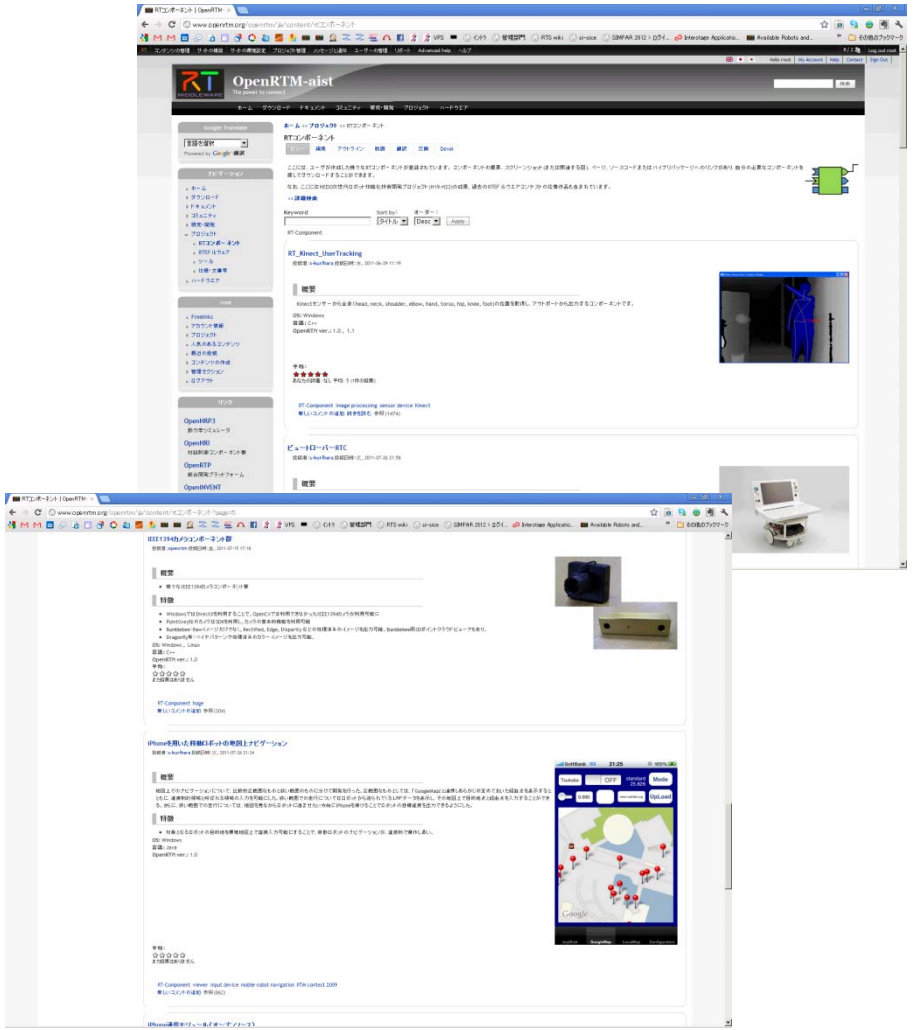
タイプ	登録数
RTコンポーネント群	287
RTミドルウエア	14
ツール	19
仕様・文書	4
ハードウエア	28

OMG RTC規格実装（11種類）

Name	Vendor	Feature
OpenRTM-aist	AIST	C++, Python, Java
OpenRTM.NET	SEC	.NET(C#,VB,C++/CLI, F#, etc..)
miniRTC, microRTC	SEC	CAN・ZigBee等を利用した組込用RTC実装
Dependable RTM	SEC/AIST	機能安全認証 (IEC61508) capableなRTM実装
RTC CANOpen	SIT, CiA	CANOpenのためのCiA (Can in automation) におけるRTC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

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を探すことができる

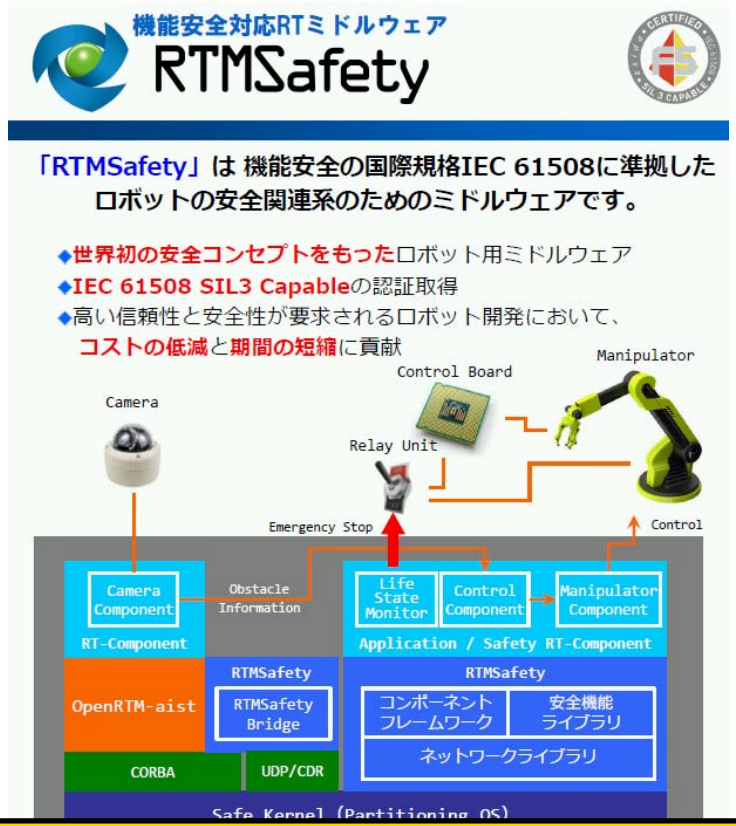


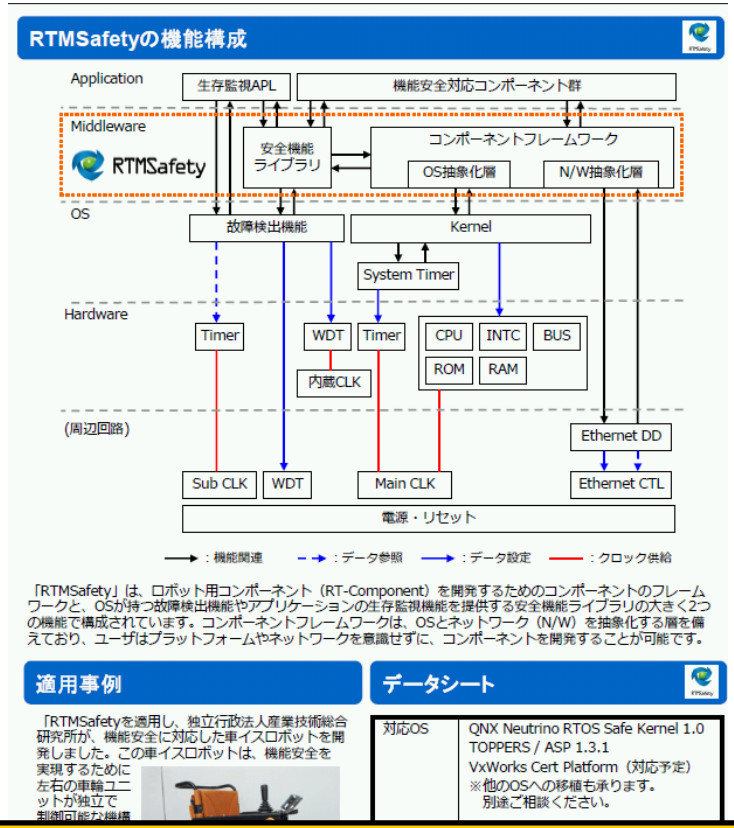
タイプ	登録数
RTコンポーネント群	287
RTミドルウエア	14
ツール	19
仕様・文書	4
ハードウエア	28

Wikiページ

- ユーザによるページ作成が行えるシステム
- Wiki（Pukiwiki）文法で簡単に作成可能
- 4つのカテゴリ
  - ノウハウ
  - ケーススタディー
  - マニュアル
  - 解説

RTMSafety





詳細は、安全・安心な RT 構築を目指して：1P1-S04（14:00-15:30）

高信頼RTミドルウエアの開発

OpenRTM-aist開発方針

- リリースの頻度を上げる
  - CI環境の拡充
- 開発効率・利用の柔軟性の向上に資する機能の追加
- 組込み機器との連携強化
- 他のミドルウエア・プラットフォームとの連携
- コンポーネント流通の仕組みの強化
- コミュニティーの交流をサポートするWeb運営

まとめ

- ロボット用ミドルウエア：OpenRTM
  - ロボットに適した共通フレームワークの提供
  - 多様な実装、多様な言語、OSに対応
- RTコンポーネントの作り方
- 1.1.0新機能
  - 多様なコールバック、新API、新EC
- 今後の展望
  - Webページを中心としたコミュニティー
  - RTMSafety

情報

- Web
  - http://openrtm.org
- RTMコンテスト
  - http://www.openrtm.org/rt/RTMcontest
- Facebook
  - http://www.facebook.com/openrtm