

ヒューマノイドロボット**HRP-4**における OpenRTM-aist応用事例

産業技術総合研究所
知能システム研究部門
ヒューマノイド研究グループ
金広 文男

独立行政法人 産業技術総合研究所

アウトライン

1. 産総研でのヒューマノイド開発の歴史
2. OpenRTMを用いたリアルタイムシステムの開発方法
 - a. リアルタイムOS
 - b. OpenRTMでリアルタイムシステムを構築するための注意事項
 - c. シミュレーションと実験との切り替え
3. **HRP-4**への適用事例

独立行政法人 産業技術総合研究所

産総研ヒューマノイドロボット開発の歴史

2000

2005

2010

HRP-2(2002)

経産省「人間協調・
共存型ロボットシス
テムの研究開発」



独立行政法人 産業技術総合研究所

可能性が検討された5つの応用分野



対人サービス



産業車両代行運転



プラント保守



ビル・ホーム管理サービス



屋外共同作業

独立行政法人 産業技術総合研究所

産総研ヒューマノイドロボット開発の歴史

2000

2005

2010

HRP-2(2002)

経産省「人間協調・
共存型ロボットシス
テムの研究開発」



HRP-3(2007)

NEDO基盤促「実環境
で働く人間型ロボット基
盤技術の研究開発」

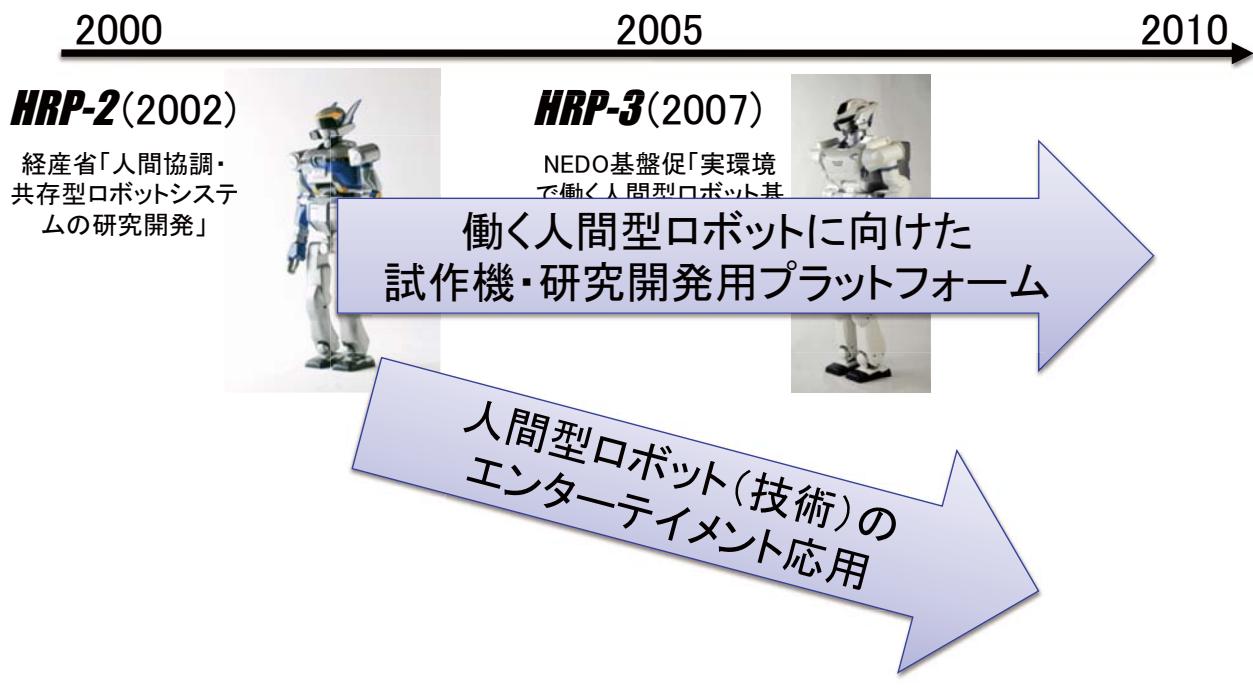


独立行政法人 産業技術総合研究所



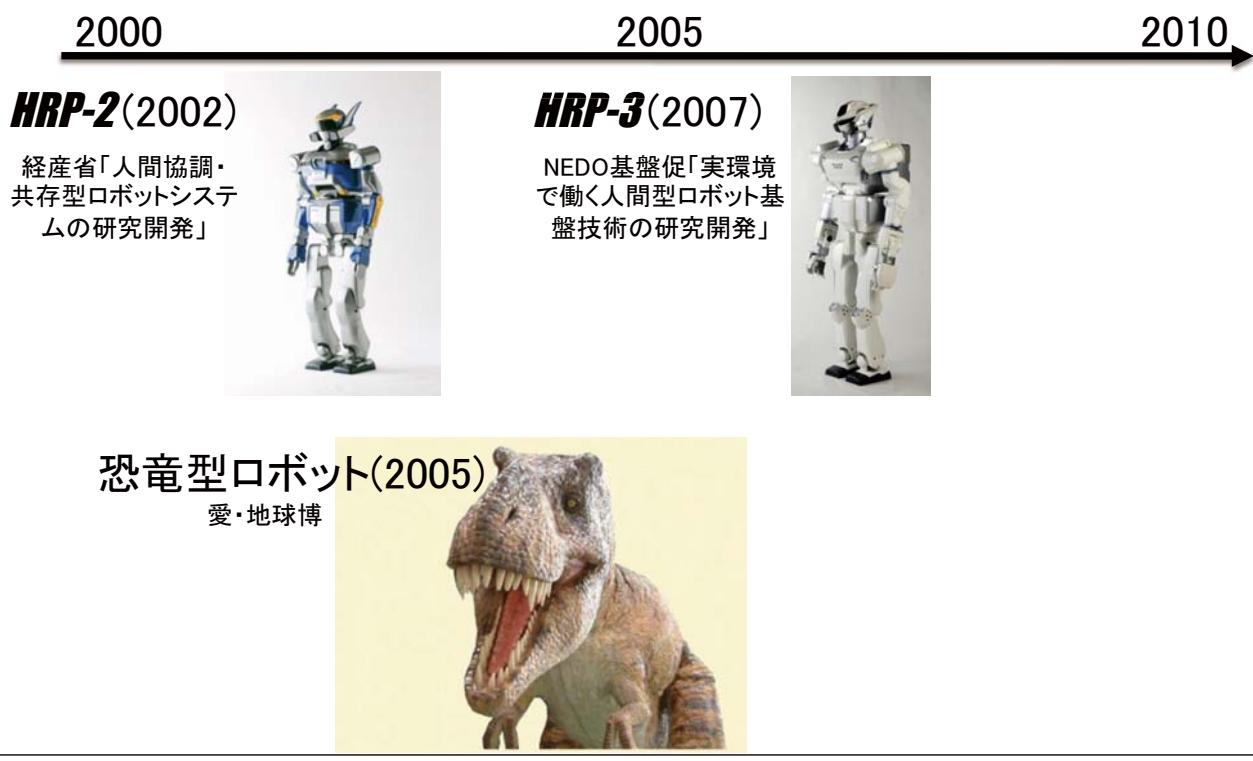
<http://robot.watch.impress.co.jp/>

産総研ヒューマノイドロボット開発の歴史



独立行政法人 産業技術総合研究所

産総研ヒューマノイドロボット開発の歴史



独立行政法人 産業技術総合研究所

TCR 02:21:04:15



産総研ヒューマノイドロボット開発の歴史

2000

2005

2010 →

HRP-2(2002)

経産省「人間協調・
共存型ロボットシス
テムの研究開発」



HRP-3(2007)

NEDO基盤促「実環境
間型ロボット基
盤技術の研究開発」



恐竜型ロボット(2005) エンターテイメント 向け 内骨格型



HRP-4C(2009) 産総研イニシアチブ “UCROA”



サイバネティックヒューマン HRP-4C



身長	158 cm
体重	43 kg(含バッテリ)
自由度 (モータの個数)	42 (顔8, 首3, 腕6×2, ハンド2×2, 腰3, 脚6×2)
バッテリ	NiMH (ニッケル水素吸蔵合金)
搭載コンピュータ	全身動作制御用: Intel Pentium M 1.6GHz 音声認識用: VIA C7 1.0GHz
搭載センサ	姿勢センサ×1 6軸力センサ×2

独立行政法人 産業技術総合研究所

新規市場創出への試み



開会挨拶

東京発 日本ファッションウィーク
シンマイ クリエーターズ コレクション
2009年3月23日



モデル

2009年ユミカツラ パリ グランド
コレクション イン 大阪
2009年7月22日



歌手

CEATEC JAPAN 2009
コンテンツ提供:ヤマハ
2009年10月6日~10日



女優

DIGITAL CONTENT EXPO
2009
コンテンツ協力:SAM
2009年10月22日~25日



MC

JISSO PROTECH 2010
ヤマハ発動機ブース
2010年6月2日~4日



歌手

CEATEC JAPAN 2010
VocalListener & VocaWatcher
2010年10月5日~9日



ダンサー

DIGITAL CONTENT EXPO 2010
コンテンツ協力:SAM,ヤマハ
2009年10月22日~25日



歓迎挨拶

APEC JAPAN 2010
JAPAN EXPERIENCE
コンテンツ協力:ヤマハ
2010年11月7日~14日

独立行政法人 産業技術総合研究所



産総研ヒューマノイドロボット開発の歴史

2000

2005

2010

HRP-2(2002)

経産省「人間協調・
共存型ロボットシステムの研究開発」



HRP-3(2007)

NEDO基盤促「実環境
で働く人間型ロボット基
盤技術の研究開発」



HRP-4(2010)

民間共同研究



恐竜型ロボット(2005)

愛・地球博



HRP-4C(2009)

産総研イニシアチブ
“UCROA”



HRP-4



独立行政法人 産業技術総合研究所

HRP-4の主な仕様

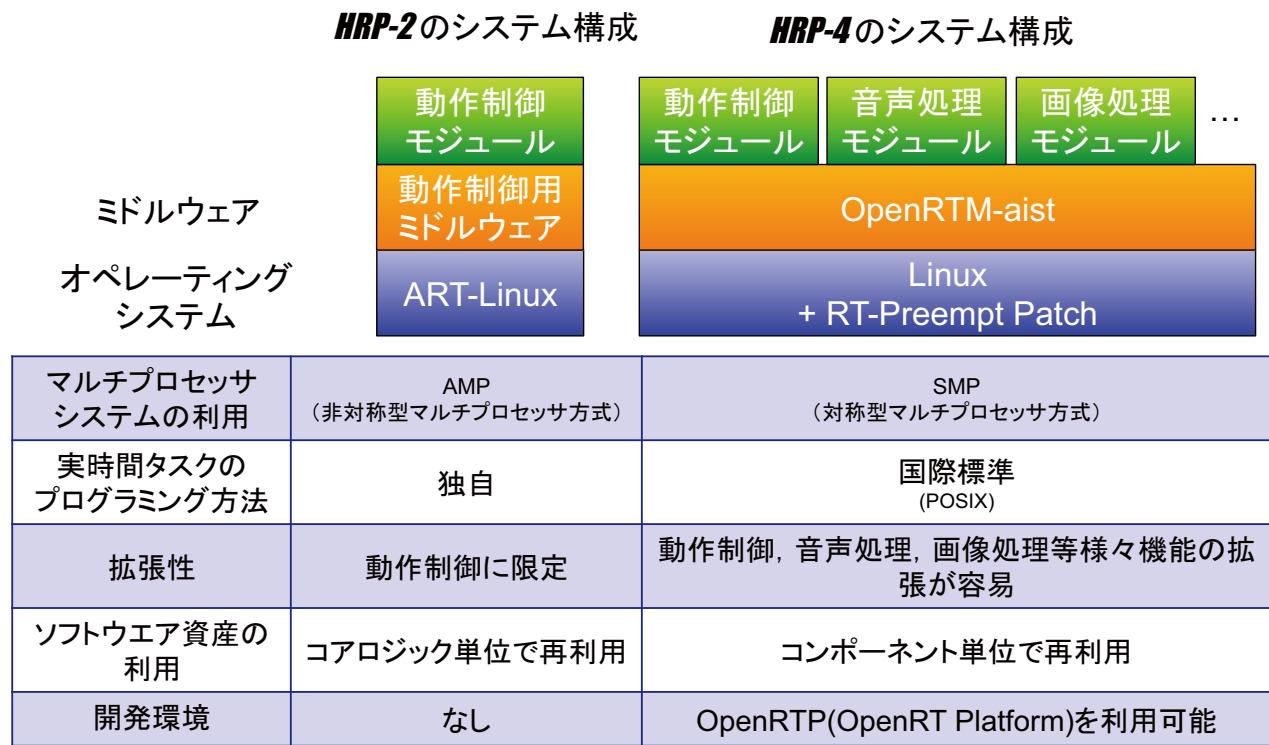


- ・ デザインコンセプト：“スリムアスリート”
- ・ 全関節に80W以下のモータを採用
- ・ 大学・研究機関向け、2,600万円

身長	151cm
体重	39kg
自由度	合計 2軸 7軸 2軸 2軸 6軸 34軸
首	2軸
腕	7軸
ハンド	2軸
腰	2軸
脚	6軸
	合計 34軸

独立行政法人 産業技術総合研究所

システム構成の比較



独立行政法人 産業技術総合研究所

リアルタイムOS(1/3)

- ・ 実時間タスク(決まった時刻までに決まった処理を完了する必要があるタスク)を実行する能力をもつオペレーティングシステム
- ・ 実世界とインタラクションするロボットの制御には不可欠
- ・ 何をやっても締切りを守ってくれる魔法のシステムではない
- ・ 処理内容が一定時間内に終わるようにリソースの配分はユーザが考える必要がある

独立行政法人 産業技術総合研究所

リアルタイムOS(2/3)

実時間(数ms程度)実行するためには…

- 大前提: 処理時間が不定となることをしない
- ディスクからの読み出し／書き込みをしない
- 標準出力／エラー出力への出力は控える
- ネットワークからの読み込み／書き出しをしない(同一ホスト内であっても)
- 不要なタスクスイッチを起こさせない

独立行政法人 産業技術総合研究所

リアルタイムOS(3/3)

- 非リアルタイムOS
 - Windows, (通常の)Linux等
- 非商用リアルタイムOS
 - ART-Linux, Preemptive Kernelなど
 - 使えるデバイス, ソフトウェア資産が豊富
- 商用
 - QNX, VxWorksなど
 - 使えるデバイスは少ないが信頼性が高い

独立行政法人 産業技術総合研究所

RT-Preemptパッチ

- ハードリアルタイム実行を可能にするための標準のLinuxカーネルに対するパッチ
- 1[ms]周期で動作するプログラムを数十[us]程度の周期誤差で実行可能
- Ubuntuの場合
apt-get install linux-rt
でインストール可
- マルチプロセッサシステムの処理能力をSMPで活用可能
- 標準規格POSIXに準拠した方法でリアルタイムで実行されるプログラムを作成可能
- 詳細はRTWiki(
https://rt.wiki.kernel.org/index.php/Main_Page)を参照

独立行政法人 産業技術総合研究所

POSIXでのプログラミング

```

param.sched_priority = MY_PRIORITY;
sched_setscheduler(0, SCHED_FIFO, &param); ← 優先度の設定

mlockall(MCL_CURRENT|MCL_FUTURE);
stack_prefault(); } ← メモリの確保

clock_gettime(CLOCK_MONOTONIC ,&t);
t.tv_sec++;

while(1) {
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL); ← 次の起床時刻まで待つ
    /* do the stuff */ ← ここで処理を行う

    t.tv_nsec += interval;
    while (t.tv_nsec >= NSEC_PER_SEC) {
        t.tv_nsec -= NSEC_PER_SEC;
        t.tv_sec++;
    }
}

```

https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO#A_Realtime_.22Hello_World.22_Example
から抜粋

独立行政法人 産業技術総合研究所

ART-Linux

- 石綿陽一氏が開発
- <http://www.dh.aist.go.jp/jp/research/assist/ART-Linux/>からダウンロード可
- タイマ割り込み(デフォルト1[ms])を全ての起点とする
- 割り込みが入った瞬間には割り込みがあったことを記録するだけ
- 割り込みハンドラは最低優先度の実時間タスクとして実行
- マルチプロセッサシステムをAMPでサポート

独立行政法人 産業技術総合研究所

ART-Linuxのプログラミング

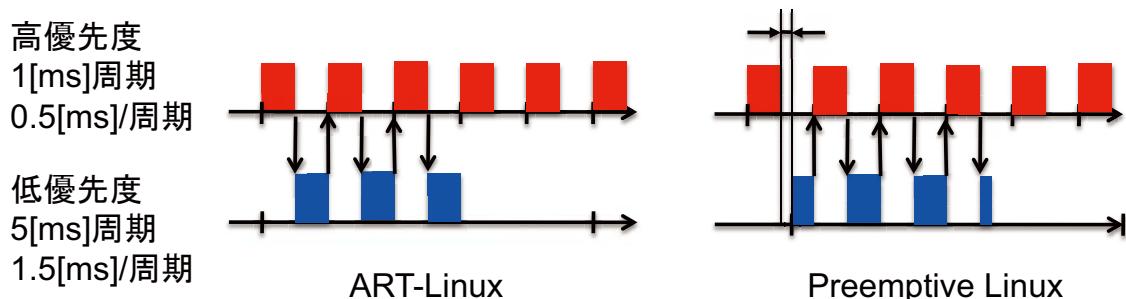
1. `art_enter`: 非実時間プロセスを実時間タスクに変換
2. `art_exit`: 実時間タスクを非実時間プロセスに変換
3. `art_wait`: 次の周期まで実行を停止
4. `art_wait_phase`
5. `art_yield`
6. `art_adjust`
7. `art_shift`

http://www.dh.aist.go.jp/jp/research/assist/ART-Linux/system_call.php

独立行政法人 産業技術総合研究所

ART-LinuxとPreemptive Linuxの違い (1/2)

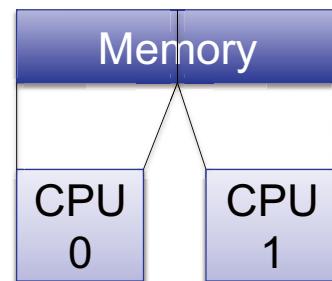
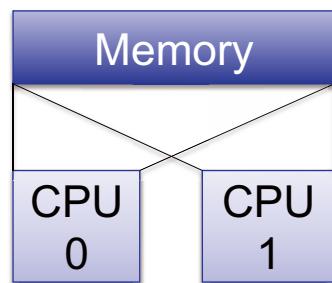
- ARTは1[ms]のタイマ割り込みを起点として動作
- Preemptive Linuxは起床時刻を絶対時刻で指定



独立行政法人 産業技術総合研究所

ART-LinuxとPreemptive Linuxの違い (2/2): SMPとAMP

- **SMP**
(Symmetric Multi Processing)
 - メモリ空間を共有
 - Windows, 通常のLinux, Preemptive Linuxはこちら
- **AMP**
(Asymmetric Multi Processing)
 - 別々のメモリ空間→物理的には1台のPCの中に論理的なPCが複数台ある状態
 - ART-Linuxはこちら



独立行政法人 産業技術総合研究所

RTCを実時間タスク化するには？

ユーザがプログラムするのは

- onActivated()
- onExecute()
- onDeactivated()
- ...

などのコールバック関数であり、それらを実行するスレッドはユーザからは見えなくなっている。
どこでart_waitやclock_nanosleepを呼び出す？

独立行政法人 産業技術総合研究所

RTコンポーネント(RTC)



独立行政法人 産業技術総合研究所

(非実時間)周期実行コンテキスト

```
// src/lib/rtm/PeriodicExecutionContext.cpp

int PeriodicExecutionContext::svc(void)
{
    RTC_TRACE(("svc()"));
    do
    {
        m_worker.mutex_.lock();
        while (!m_worker.running_)
        {
            m_worker.cond_.wait();
        }
        if (m_worker.running_)
        {
            std::for_each(m_comps.begin(), m_comps.end(), invoke_worker());
        }
        m_worker.mutex_.unlock();
        if (!m_nowait) { coil::sleep(m_period); }
    } while (m_svc);
    return 0;
}
```

実行周期分待つ

実行コンテキストのメイン部分
このメンバ関数がスレッドで実行される

コンポーネントの状態遷移マシンを駆動

独立行政法人 産業技術総合研究所

実時間周期実行コンテキスト

```
// src/ext/artlinux/art_ec/ArtExecutionContext.cpp

int ArtExecutionContext::svc(void)
{
    if (art_enter(ART_PRIO_MAX, ART_TASK_PERIODIC, m_usec) == -1)
    {
        std::cerr << "fatal error: art_enter" << std::endl;
    }
    do
    {
        std::for_each(m_comps.begin(), m_comps.end(), invoke_worker());
        if (art_wait() == -1)
        {
            std::cerr << "fatal error: art_wait" << std::endl;
        }
    } while (m_running);
    if (art_exit() == -1)
    {
        std::cerr << "fatal error: art_exit" << std::endl;
    }
    return 0;
}
```

次の実行周期を待つ

実時間タスクに変換

コンポーネントの状態遷移マシンを駆動

独立行政法人 産業技術総合研究所

実行コンテキストの切り替え

rtc.conf

```
corba.nameservers: localhost:2809
naming.formats: %n.rtc
logger.enable: YES
logger.log_level: NORMAL
logger.file_name: stdout
exec_ctxt.periodic.rate: 1000
```

```
manager.modules.load_path: /usr/local/lib
manager.modules.preload: ArtExecutionContext.so
exec_ctxt.periodic.type: ArtExecutionContext
```

シェアードオブジェクト
を検索するパス

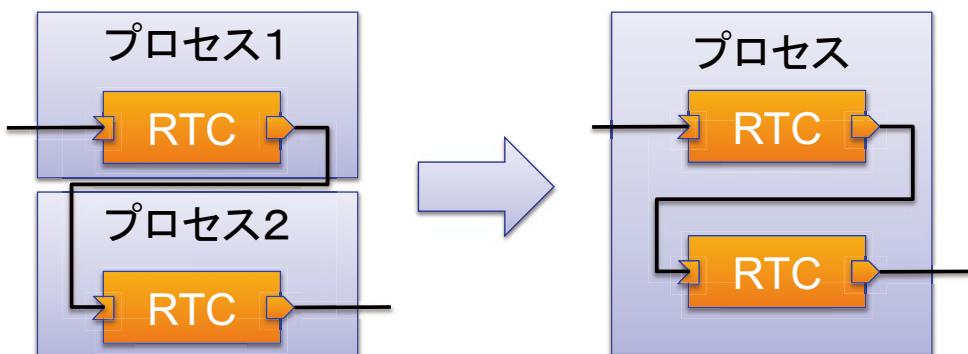
シェアードオブジェクト
形式の実行コンテキス
トファクトリをロード

実行コンテキストのタイプを指定

独立行政法人 産業技術総合研究所

複数のRTCを連携させながら実時間実行する には？

1. 個々のRTCを実時間実行コンテキストで実行しても、
RTC間の通信が実時間性を阻害してしまう
 - 同一プロセス上に複数のRTCを生成して実行することで、
プロセス間通信を不要にする(多くのCORBA実装では
同一プロセス内の通信は関数呼び出しに置き換えられ
る)



独立行政法人 産業技術総合研究所

RTC生成処理の流れ(1/3)

// examples/SimpleIO/ConsoleInComp.cpp

```
int main (int argc, char** argv)
{
    RTC::Manager* manager;
    manager = RTC::Manager::init(argc, argv); // マネージャの初期化  

    // マネージャは1つのプロセス  

    // には1つだけ存在する

    // Set module initialization procedure
    // This procedure will be invoked in activateManager() function.
    manager->setModuleInitProc(MyModuleInit); // コンポーネントのファクトリを  

                                                // 登録し、インスタンスを生成  

                                                // する関数を登録  

                                                // (activateManager()から呼  

                                                // ばれる)

    // Activate manager and register to naming service
    manager->activateManager(); // コンポーネントのファクトリを  

                                // 登録し、インスタンスを生成  

                                // する関数を登録  

                                // (activateManager()から呼  

                                // ばれる)

    // run the manager in blocking mode
    // runManager(false) is the default.
    manager->runManager();

    // If you want to run the manager in non-blocking mode, do like this
    // manager->runManager(true);

    return 0;
}
```

マネージャの初期化
マネージャは1つのプロセス
には1つだけ存在する

コンポーネントのファクトリを
登録し、インスタンスを生成
する関数を登録
(activateManager()から呼
ばれる)

独立行政法人 産業技術総合研究所

RTC生成処理の流れ(2/3)

// examples/SimpleIO/ConsoleInComp.cpp

```
void MyModuleInit(RTC::Manager* manager)
{
    ConsoleInInit(manager); // コンポーネントのファクトリを  

                            // マネージャに登録

    RTC::RtcBase* comp;

    // Create a component
    std::cout << "Creating a component: \"ConsoleIn\"....";
    comp = manager->createComponent("ConsoleIn");
    std::cout << "succeed." << std::endl;
```

“ConsoleIn”というコンポー
ネントファクトリを使ってコン
ポーネントのインスタンスを
生成

// examples/SimpleIO/ConsoleIn.cpp

```
void ConsoleInInit(RTC::Manager* manager)
{
    RTC::Properties profile(consolein_spec);
    manager->registerFactory(profile,
        RTC::Create<ConsoleIn>,
        RTC::Delete<ConsoleIn>);
```

コンポーネントのファクトリを
マネージャに登録

独立行政法人 産業技術総合研究所

RTC生成処理の流れ(3/3)

```
// src/lib/rtm/Manager.cpp
RTObject_impl* Manager::createComponent(const char* comp_args){
    ...
    FactoryBase* factory(m_factory.find(comp_id)); ← コンポーネントファクトリを検索
    ...
    RTObject_impl* comp;
    comp = factory->create(this); ← コンポーネントのインスタンスを生成
    ...
    comp->initialize()
    ...
}

// src/lib/rtm/RTObject.cpp
ReturnCode_t RTObject_impl::initialize(){
    ...
    RTC::ExecutionContextBase* ec;
    ec = RTC::Manager::instance().createContext(ec_args.c_str()); ← 実行コンテキストのインスタンスを生成
    ...
    ec->bindComponent(this); ← コンポーネントを実行コンテキストに登録
    ...
    ReturnCode_t ret;
    ret = on_initialize();
    ...
    return ret;
}
```

独立行政法人 産業技術総合研究所

RTC生成処理の流れのまとめ

1. Managerを初期化
2. Managerにコンポーネントファクトリを登録
3. コンポーネントファクトリからコンポーネントを生成
4. 実行コンテキストファクトリから実行コンテキストを生成
5. 実行コンテキストにコンポーネントを登録

rtcd

// utils/rtcd/rtcd.cpp

```
int main (int argc, char** argv)
{
    RTC::Manager* manager;
    manager = RTC::Manager::init(argc, argv);

    manager->activateManager();

    manager->runManager();

    return 0;
}
```

マネージャだけ生成している

独立行政法人 産業技術総合研究所

rtcdを使って複数コンポーネントを同一プロセス 上に生成する方法1:rtc.confを利用

rtc.conf

```
corba.nameservers: localhost:2809
naming.formats: %n.rtc
logger.enable: YES
logger.log_level: NORMAL
logger.file_name: stdout
exec_ctxt.periodic.rate: 1000
```

```
manager.modules.load_path: /usr/local/lib
manager.modules.preload: componentA.so, componentB.so
manager.components.precreate: componentA, componentB
```

シェアードオブジェクト
を検索するパス

シェアードオブジェクト
形式のコンポーネント
ファクトリをロード

コンポーネントの生成を指示

独立行政法人 産業技術総合研究所

rtcdを使って複数コンポーネントを同一プロセス上に生成する方法2:マネージャのインターフェースを利用

```
// Manager.idl

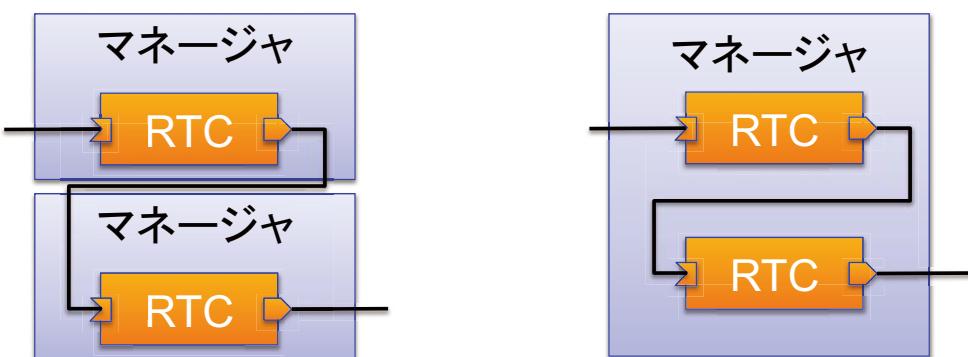
interface Manager
{
    RTC::ReturnCode_t load_module(in string pathname, in string initfunc);
    RTC::RTObject create_component(in string module_name);
    ...
};
```

シェアードオブジェクト形式のコンポーネントファクトリや実行コンテキストファクトリをダイナミックロード

ロードしたコンポーネントファクトリからインスタンスを生成

独立行政法人 産業技術総合研究所

マネージャとRTC



- RTCが異常終了しても他の部分は動き続ける
- RTC間の通信にはプロセス間通信が発生
- 1つのRTCが異常終了すると全体が落ちる
- RTC間のプロセス間通信がなくなり、実時間実行が可能に

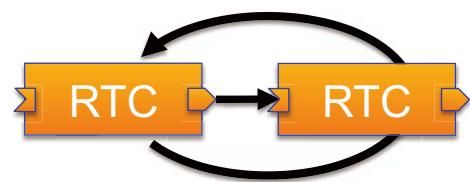
独立行政法人 産業技術総合研究所

複数のRTCを連携させながら実時間実行するには？

1. 個々のRTCを実時間実行コンテキストで実行しても、RTC間の通信が実時間性を阻害してしまう
 - 同一プロセス上に複数のRTCを生成して実行することで、プロセス間通信を不要にする(多くのCORBA実装では同一プロセス内の通信は関数呼び出しに置き換えられる)
2. データの依存関係に基づいてコンポーネントを特定の順序で実行する必要がある
 - 複数のコンポーネントを1つの実行コンテキストで駆動して実行順序を制御

独立行政法人 産業技術総合研究所

実行コンテキストとRTCの組み合わせ



独立行政法人 産業技術総合研究所

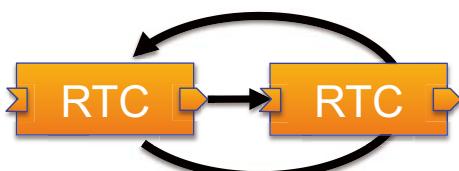
RTCとECの組み合わせを変更する方法

// RTC.idl

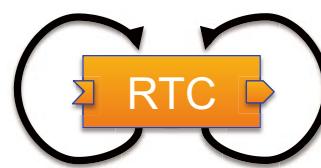
```
interface ExecutionContext
{
    ...
    ReturnCode_t add_component(in LightweightRTObject comp);
    ReturnCode_t remove_component(in LightweightRTObject comp);
    ...
}
```

compをこのECで実行してもらう

compをこのECで実行するのをやめる



ec.add_component(rtc1);
ec.add_component(rtc2);



ec1.add_component(rtc);
ec2.add_component(rtc);

独立行政法人 産業技術総合研究所

複数のRTCを連携させながら実時間実行するには？

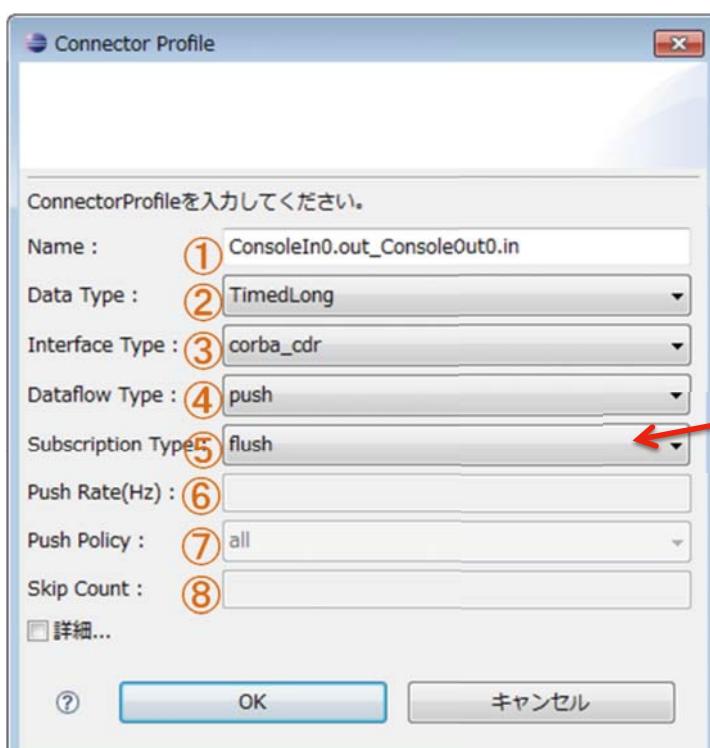
1. 個々のRTCを実時間実行コンテキストで実行しても、RTC間の通信が実時間性を阻害してしまう
 - 同一プロセス上に複数のRTCを生成して実行することで、プロセス間通信を不要にする(多くのCORBA実装では同一プロセス内の通信は関数呼び出しに置き換えられる)
2. データの依存関係に基づいてコンポーネントを特定の順序で実行する必要がある
 - 複数のコンポーネントを1つの実行コンテキストで駆動して実行順序を制御
 - ポート間接続のサブスクリプションタイプ、バッファサイズを適切に設定

サブスクリプションタイプ

- New
 - 送信バッファにデータが入り次第送信スレッドが送信を開始する
- Periodic
 - 送信バッファにたまつたデータを送信スレッドが定期的に送信する
- Flush
 - write()を呼び出したスレッドが実際の送信を実行する。write()は送信が完了するまで戻らない

ECとは別のスレッドが介在
↓
挙動の予測が困難に

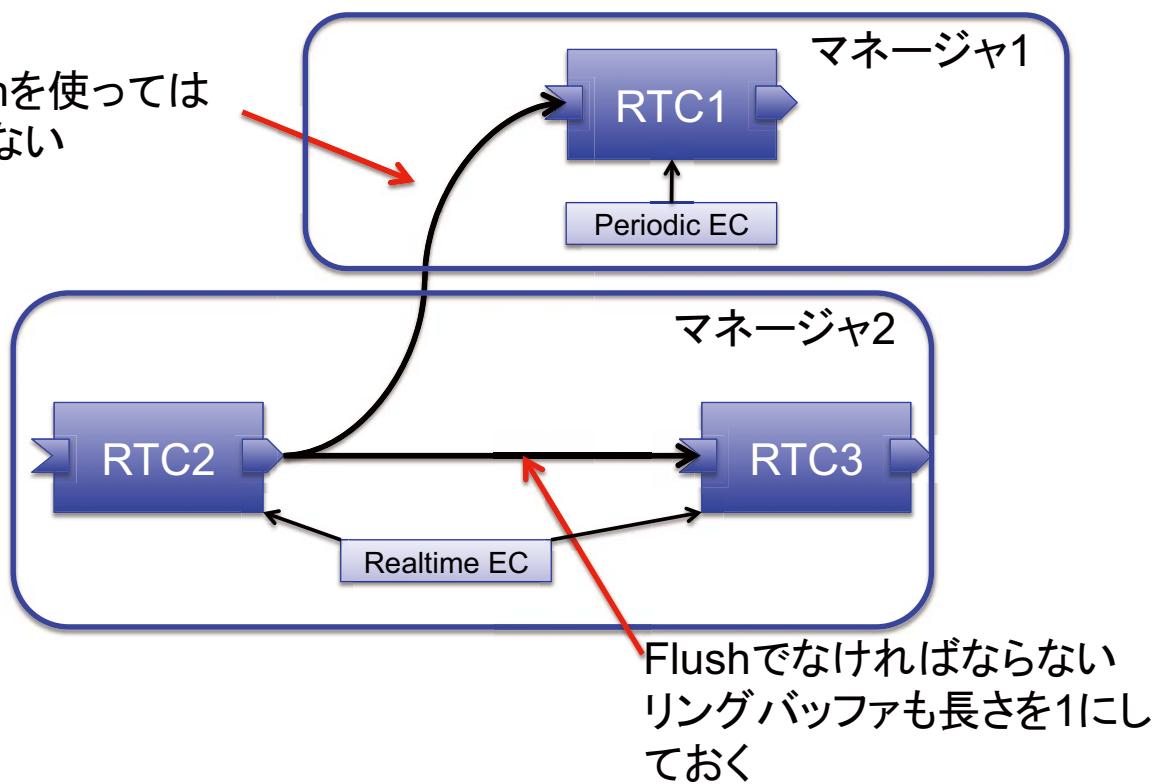
サブスクリプションタイプの設定



サブスクリプション
タイプの設定

サブスクリプションタイプの選択

Flushを使っては
いけない



独立行政法人 産業技術総合研究所

その他のTips

- ログレベルに注意(LoggerをDisableしていても)

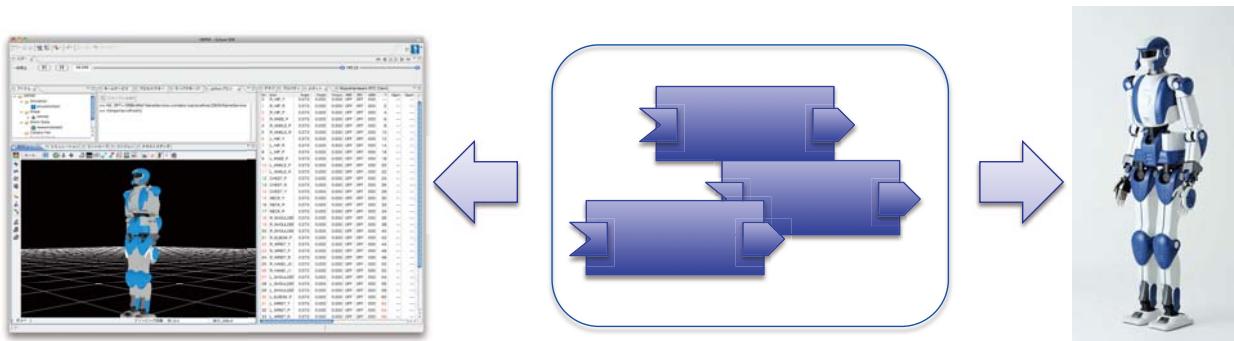
```
# rtc.conf
corba.nameservers: localhost:2809
naming.formats: %n.rtc
logger.enable: NO
logger.log_level: PARANOIA
```

```
# rtc.conf
corba.nameservers: localhost:2809
naming.formats: %n.rtc
logger.enable: YES
logger.log_level: NORMAL
```

シミュレーションと実験との切り替え

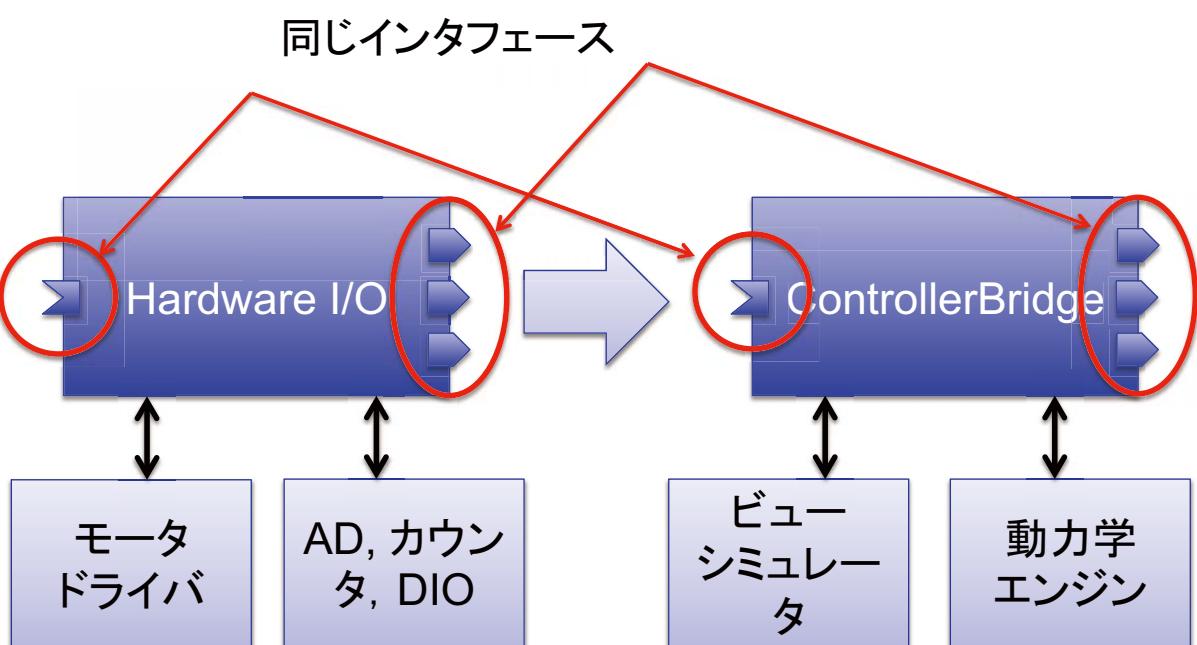
1. 入出力先が異なる

- シミュレーション時はシミュレーション世界の中に存在するロボット、実験時は実際のロボット
→入出力先コンポーネントのインターフェースを合わせておいて切り替える



独立行政法人 産業技術総合研究所

シミュレータと実機の切り替え



独立行政法人 産業技術総合研究所

シミュレーションと実験との切り替え

1. 入出力先が異なる

- シミュレーション時はシミュレーション世界の中に存在するロボット、実験時は実際のロボット
→入出力先コンポーネントのインターフェースを合わせておいて切り替える

2. 時間の進み方が異なる

- コントローラは実世界の時間、シミュレータはシミュレーション時間で動く
→コントローラを実世界の時間で動かす代わりに、シミュレーションループの実行に同期して動かす

独立行政法人 産業技術総合研究所

シミュレーション用実行コンテキスト



独立行政法人 産業技術総合研究所

外部トリガ駆動実行コンテキスト

// src/lib/rtm/OpenHRPExecutionContext.cpp

```
int OpenHRPExecutionContext::svc(void)
{
    return 0;
}
```

外部トリガ駆動実行コンテキストのインターフェース定義

メインではなにもしない

// src/lib/rtm/idl/OpenRTM.idl

```
interface ExtTrigExecutionContextService
: RTC::ExecutionContextService
{
    void tick();
};
```

```
void OpenHRPExecutionContext::tick()
throw (CORBA::SystemException)
{
    std::for_each(m_comps.begin(), m_comps.end(), invoke_worker());
    return;
}
```

外部からのトリガで状態遷移マシンを駆動する

独立行政法人 産業技術総合研究所

HRP-4 デモの内容

- 運動機能
 - 歩行, 体操, ポーズ
- 音声機能
 - 発話
 - 音声認識結果をトリガとした動作の実行
- 視覚機能
 - 顔認識+追従動作
 - 色認識+追従動作



独立行政法人 産業技術総合研究所

4台の計算機を使用

音声処理兼操作PC(外部)
Ubuntu10.04
Core2Duo 2.4G
OpenRTM-aist-Python, Java



画像処理PC(背中に搭載)
Windows XP
Atom 1.33G
OpenRTM-aist-C++



画像表示PC
(外部)
Windows XP
PentiumM
1.1G
OpenRTM-aist-C++



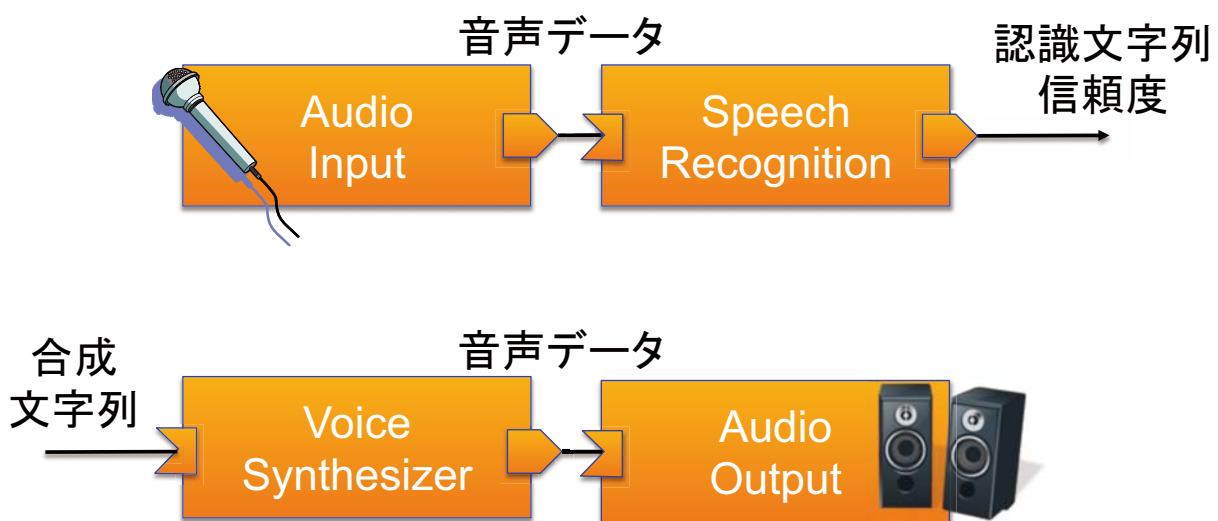
運動制御PC(体内)

Ubuntu8.04+Preemptive Kernel
Pentium M1.6G
OpenRTM-aist-C++



独立行政法人 産業技術総合研究所

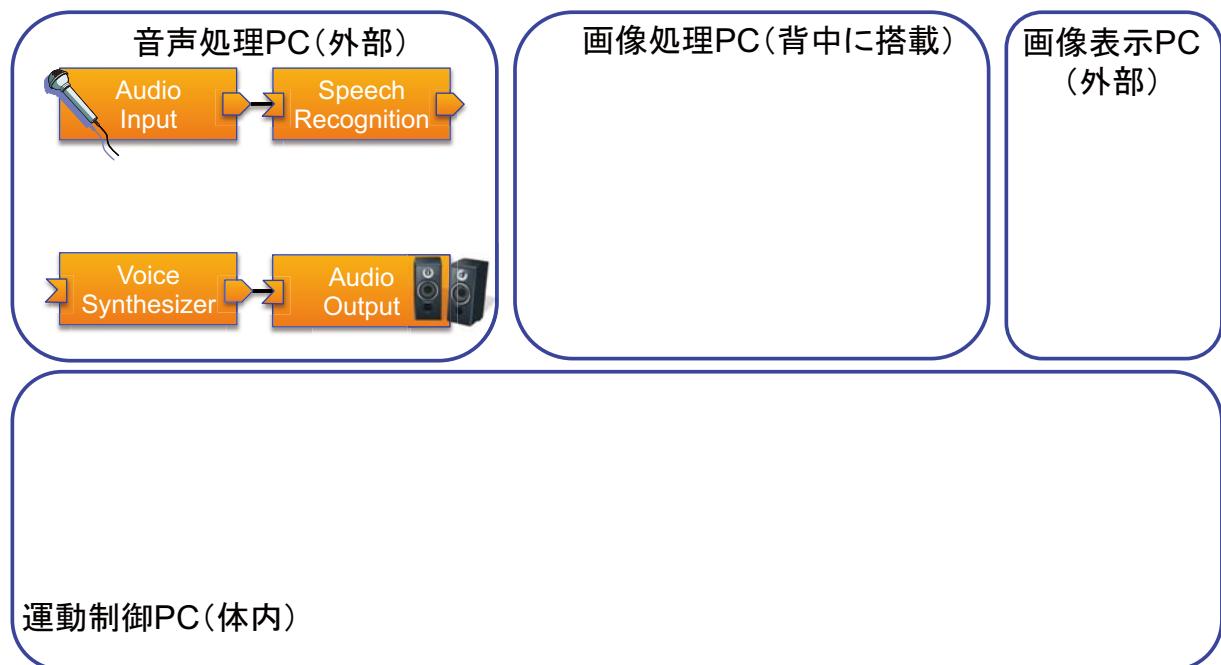
音声処理サブシステム



OpenHRI(<http://openhri.net/>)のRTCを利用

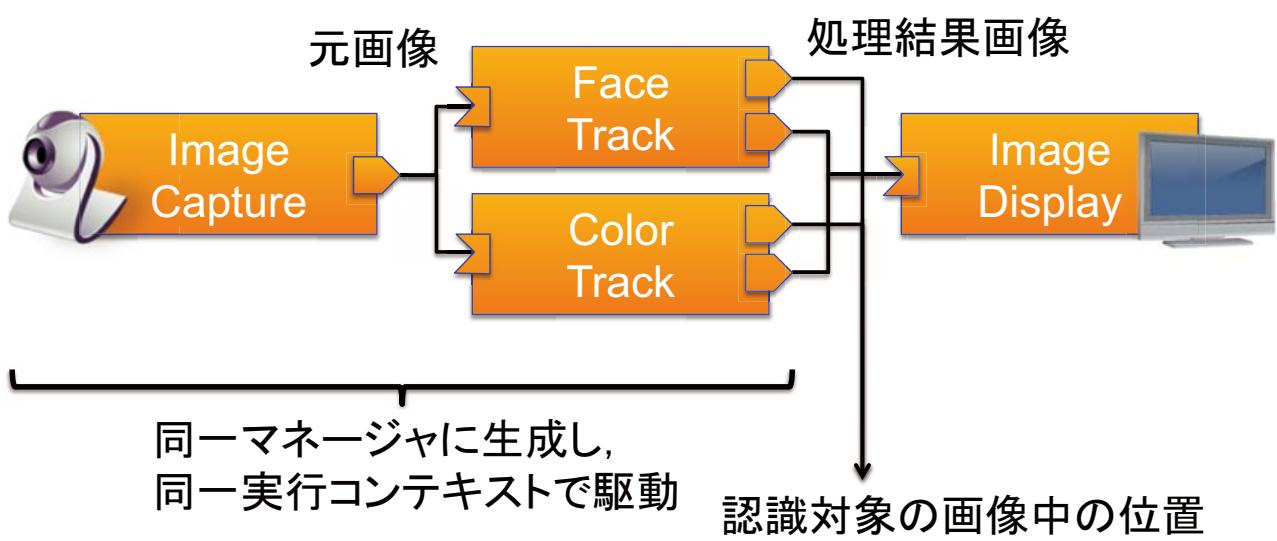
独立行政法人 産業技術総合研究所

音声関連RTC群



独立行政法人 産業技術総合研究所

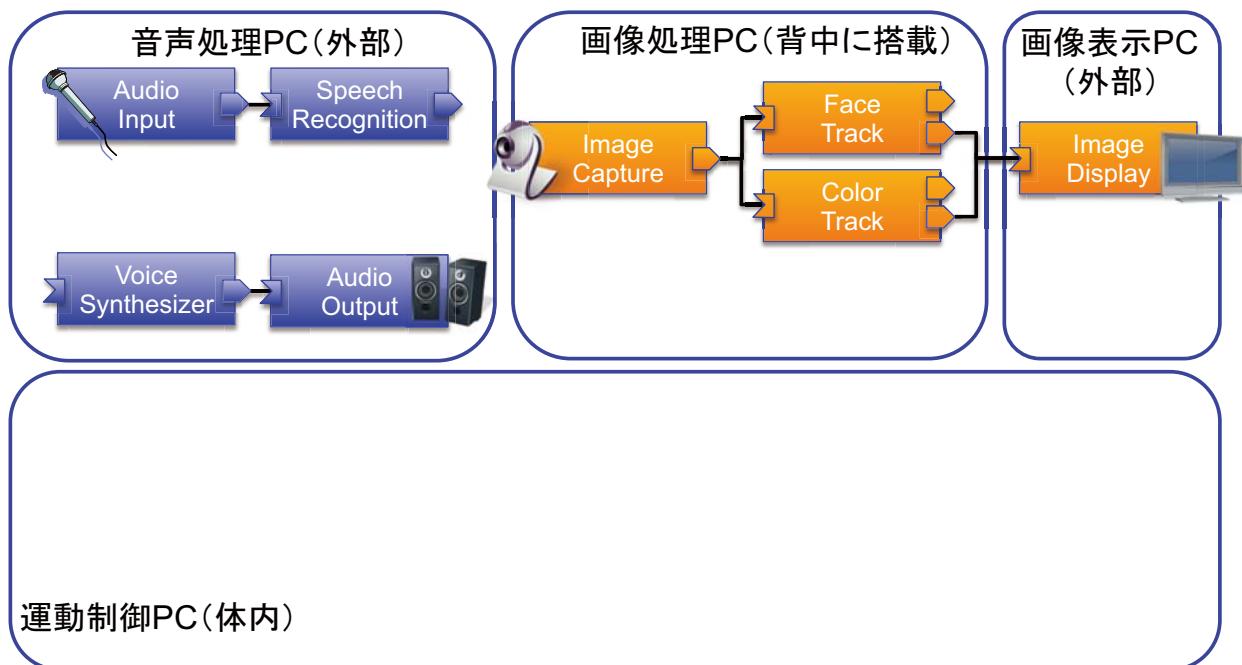
画像処理サブシステム



OpenCVでコアロジックを実装

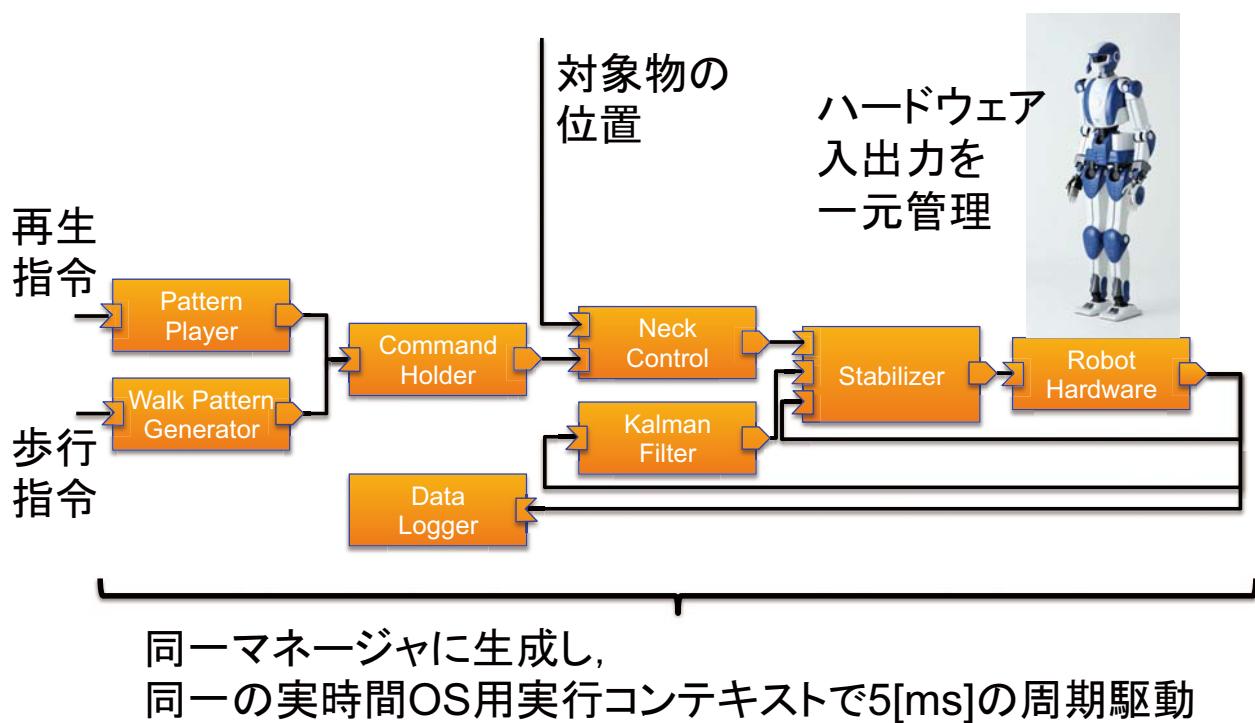
独立行政法人 産業技術総合研究所

視覚関連RTC群



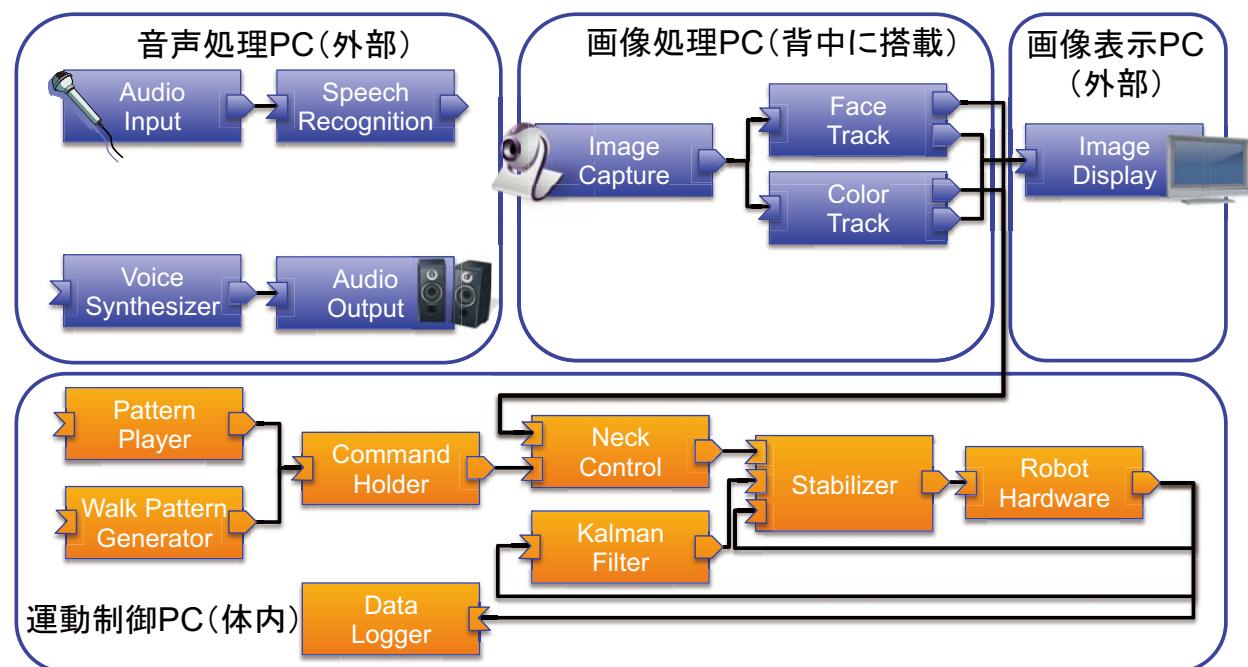
独立行政法人 産業技術総合研究所

運動制御サブシステム



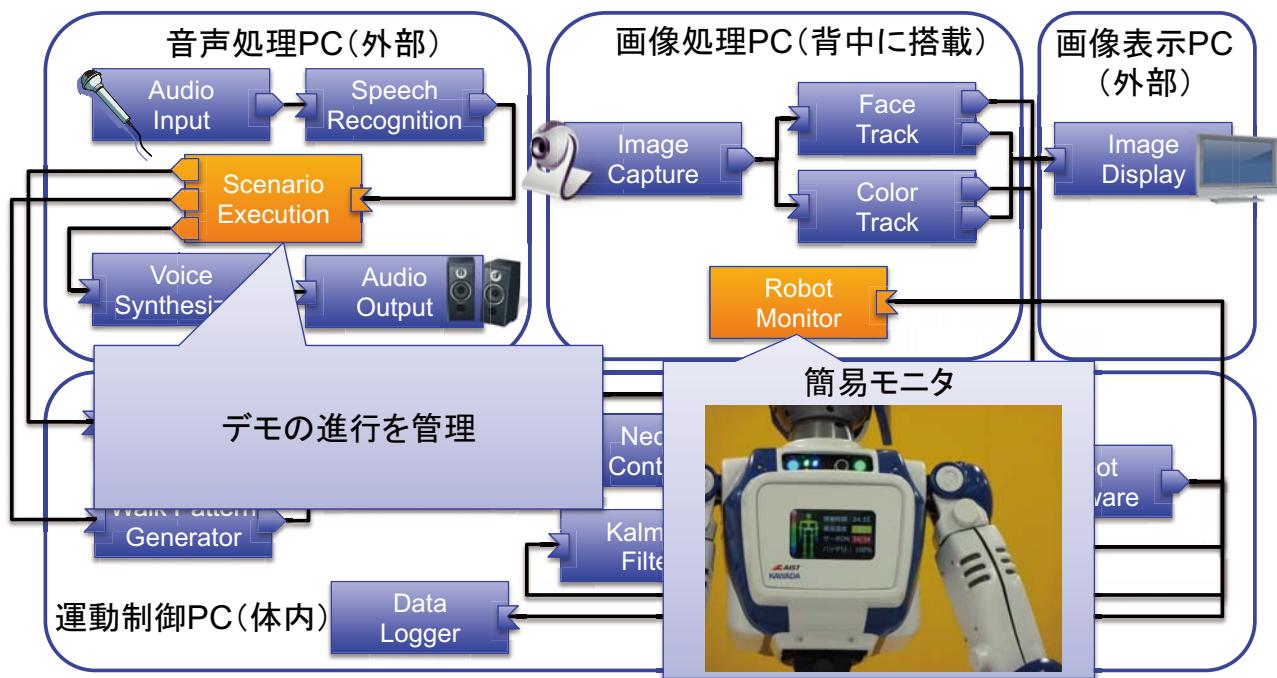
独立行政法人 産業技術総合研究所

運動関連RTC群



独立行政法人 産業技術総合研究所

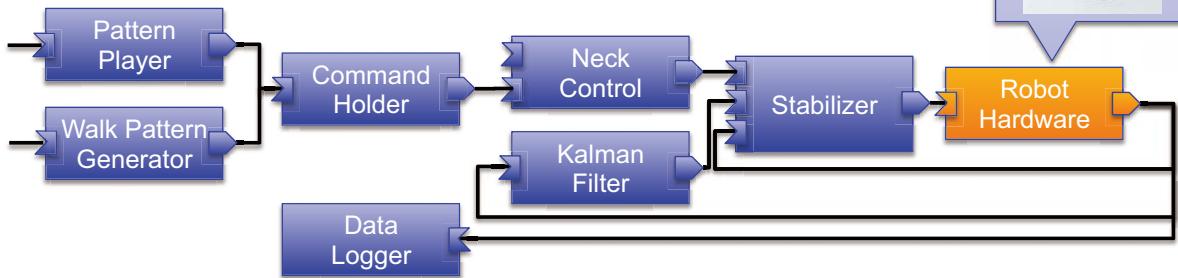
その他



独立行政法人 産業技術総合研究所

実験時の構成

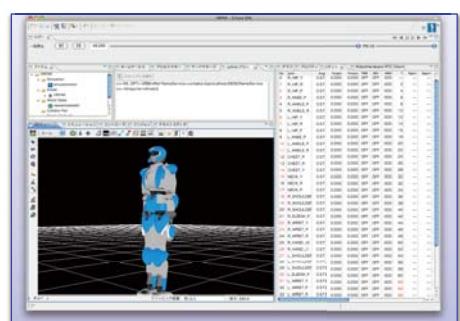
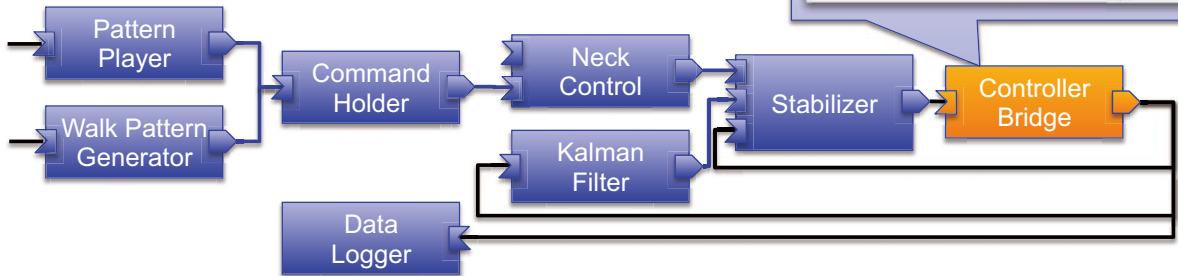
- ・ハードウェア入出力を一元管理
- ・リアルタイムOSの機能を利用してRTC群を周期実行



独立行政法人 産業技術総合研究所

シミュレーション時の構成

- ・シミュレータへの入出力を一元管理
- ・シミュレーション世界の時刻に従ってRTC群を駆動



独立行政法人 産業技術総合研究所

まとめ

- OpenRTM-aistを用いたリアルタイムシステムの構築方法
 - 実時間実行コンテキストの利用
 - 1つのマネージャ上に複数のRTCを生成して通信処理を削除
 - 実行コンテキストとRTCの対応関係, サブスクリプションタイプ, バッファサイズを適切に設定
- ヒューマノイドロボット**HRP-4**への適用事例を紹介
 - 実時間(5[ms]周期実行), 非実時間のタスクが混在
 - 4台の計算機に分散した20個程度のRTCからなるシステム

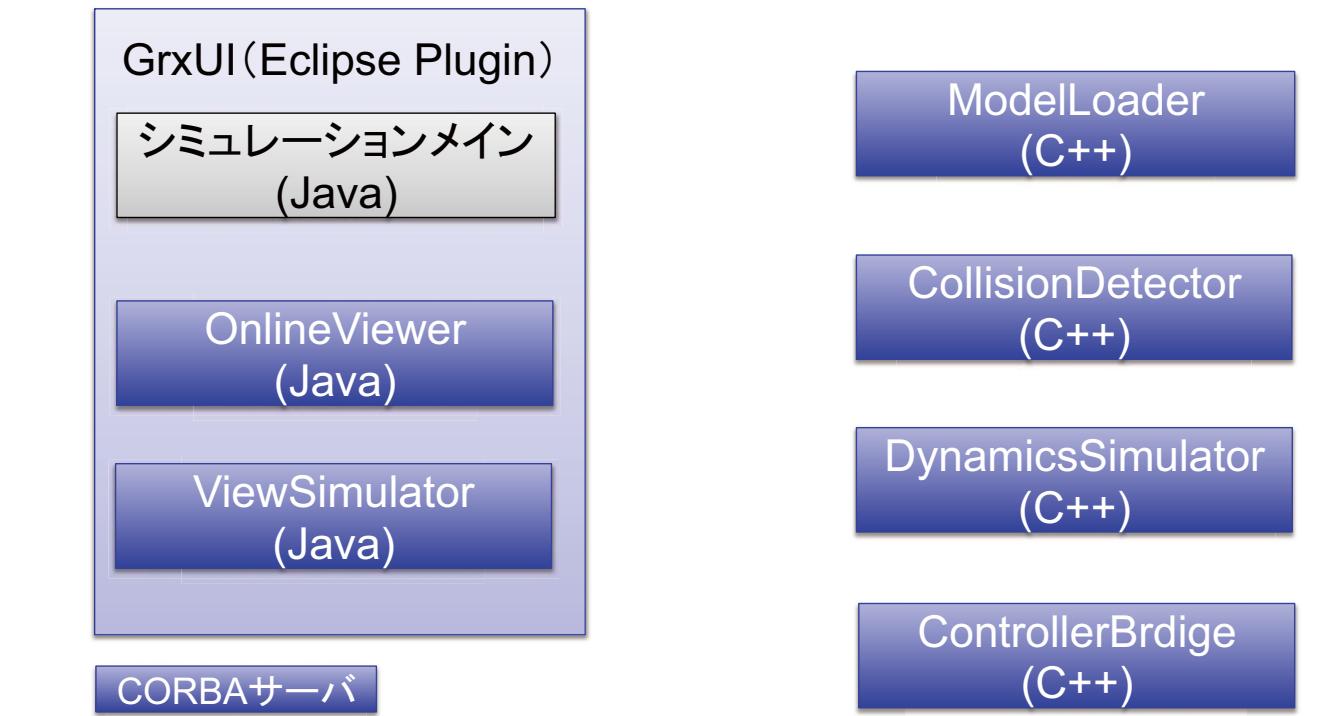
独立行政法人 産業技術総合研究所

おまけ:OpenHRPのちょっと変わった利用方法

1. OpenHRPの内部構造
2. スクリプトでシミュレーションを繰り返し実行する
3. シミュレーションのメインループを自分で記述する
4. GrxUIをビューアとして利用する
5. 計算ライブラリとして利用する
6. GrxUIをロボットのモニタとして利用する

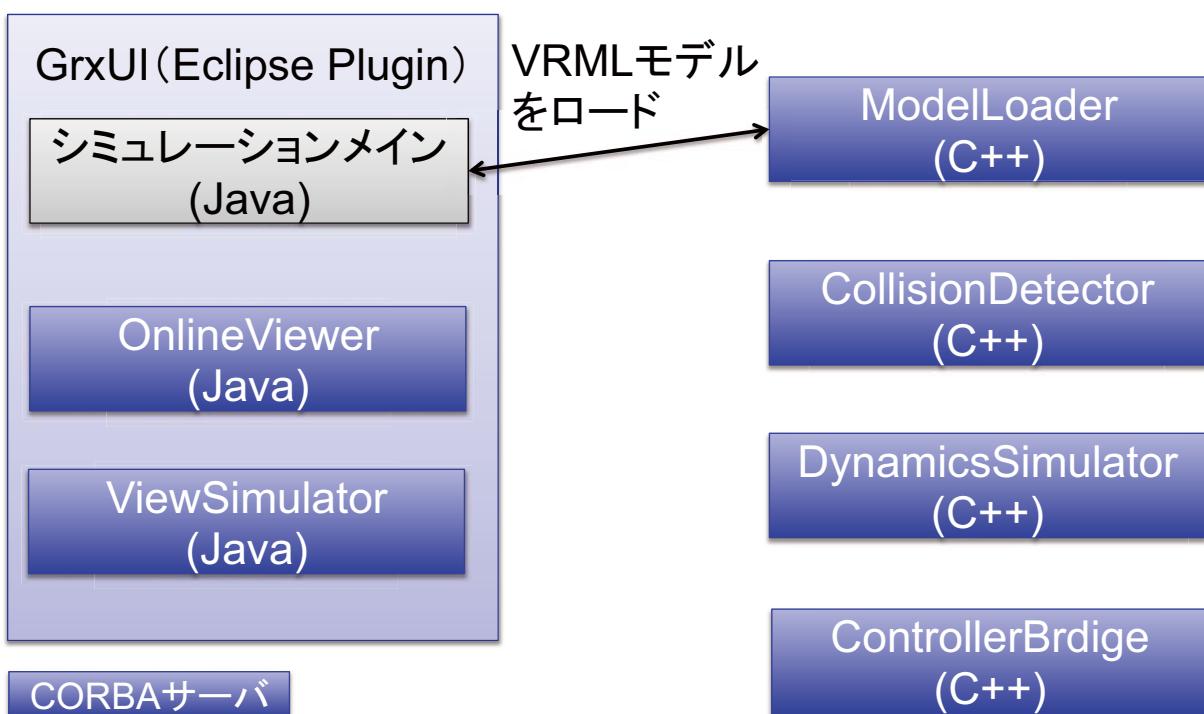
独立行政法人 産業技術総合研究所

OpenHRPの内部構造



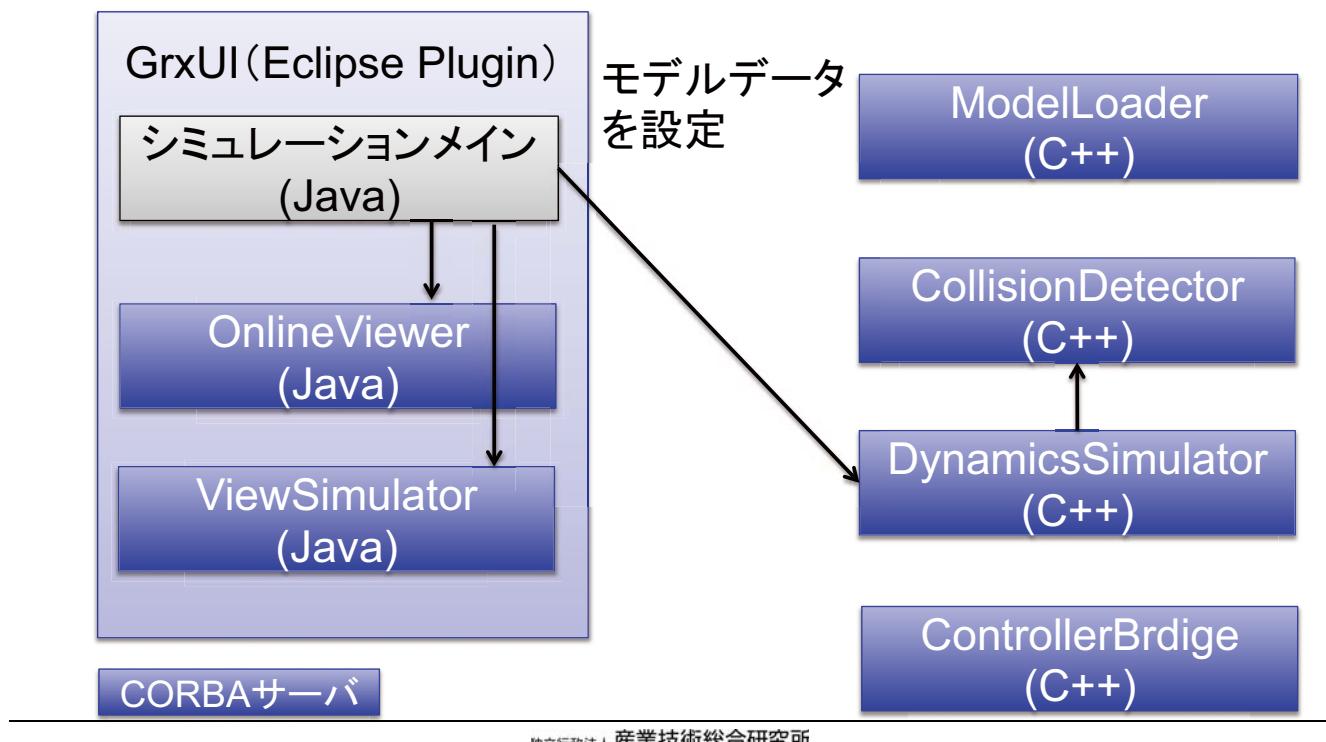
独立行政法人 産業技術総合研究所

OpenHRPの内部構造: 初期化(1/2)



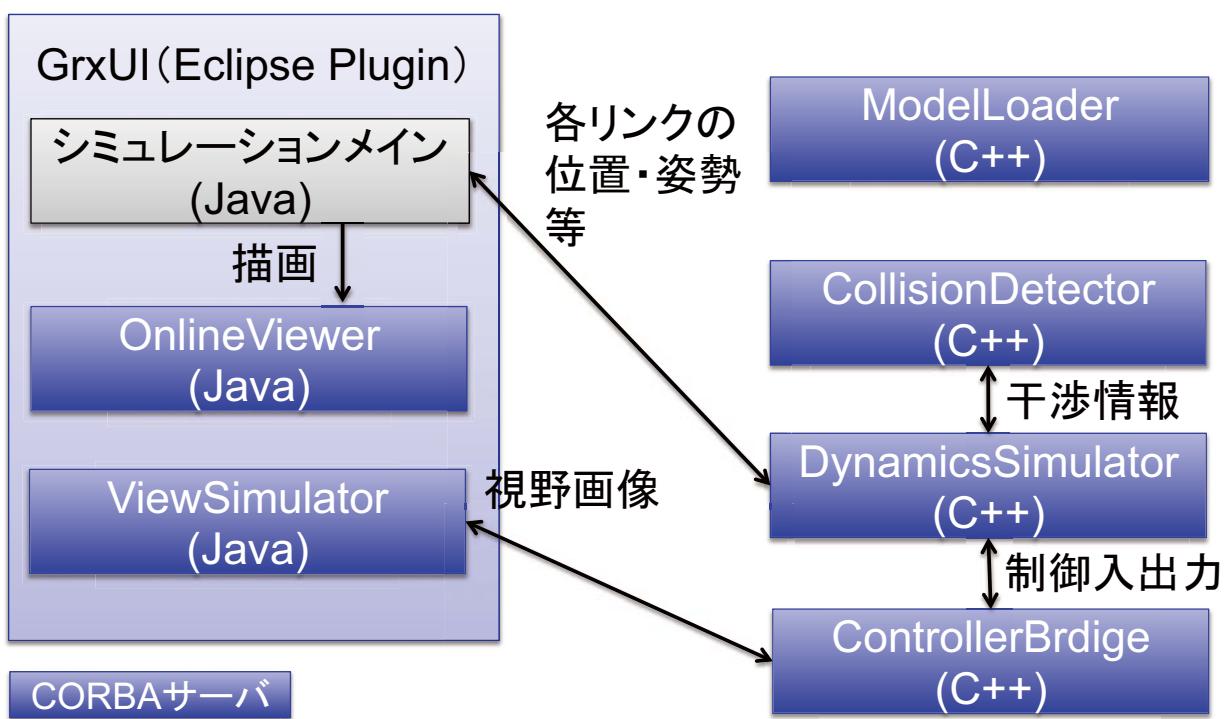
独立行政法人 産業技術総合研究所

OpenHRPの内部構造: 初期化(2/2)



独立行政法人 産業技術総合研究所

OpenHRPの内部構造: メインループ



独立行政法人 産業技術総合研究所

スクリプトでシミュレーションを繰り返し実行する

```

import time
import com.generalrobotix.ui.item.GrxFolderItem as GrxFolderItem
import syncExec
import java.lang.Runnable as Runnable

class StartSim(Runnable):
    def run(self):
        sim.startSimulation(0)
        return None

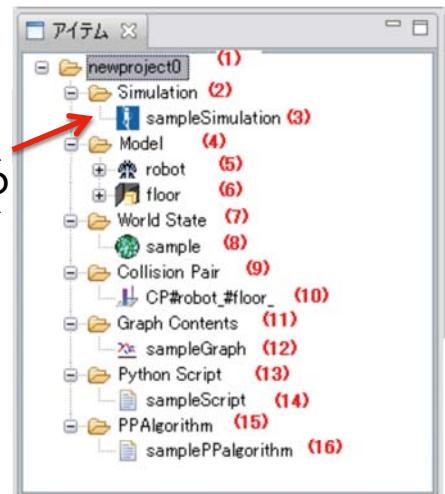
class SetTime(Runnable):
    def run(self):
        sim.setDbl("totalTime", 5.0)
        return None

sim = uimanager.getSelectedItem(GrxFolderItem, None)
syncExec.execute(SetTime())

for i in range(3):
    syncExec.execute(StartSim())
    sim.waitStopSimulation()

```

現在選択されている
シミュレーションアイ
テムを取得



シミュレーションを開始
シミュレーションの終了を待つ

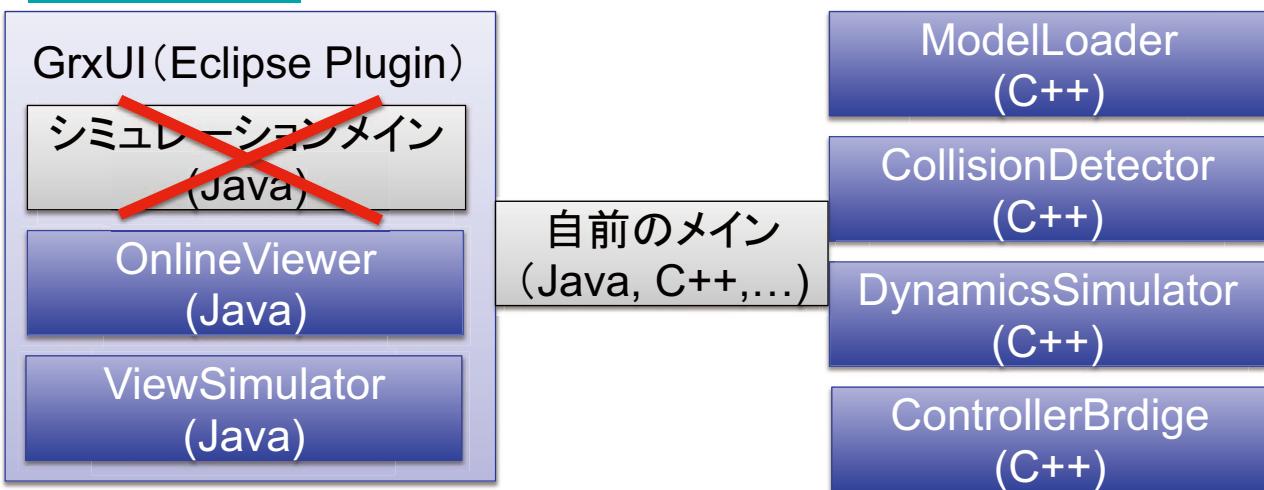
独立行政法人 産業技術総合研究所

シミュレーションのメインループを自分で記述する

例：

OpenHRP3/sample/example/scheduler/scheduler.cpp(付属)

[http://code.google.com/p/rtm-ros-robotics/source/
browse/trunk/rtmros_common/openhrp3/src/openhrp-scheduler.cpp](http://code.google.com/p/rtm-ros-robotics/source/browse/trunk/rtmros_common/openhrp3/src/openhrp-scheduler.cpp)(岡田先生)



独立行政法人 産業技術総合研究所

計算ライブラリとして利用する

例:

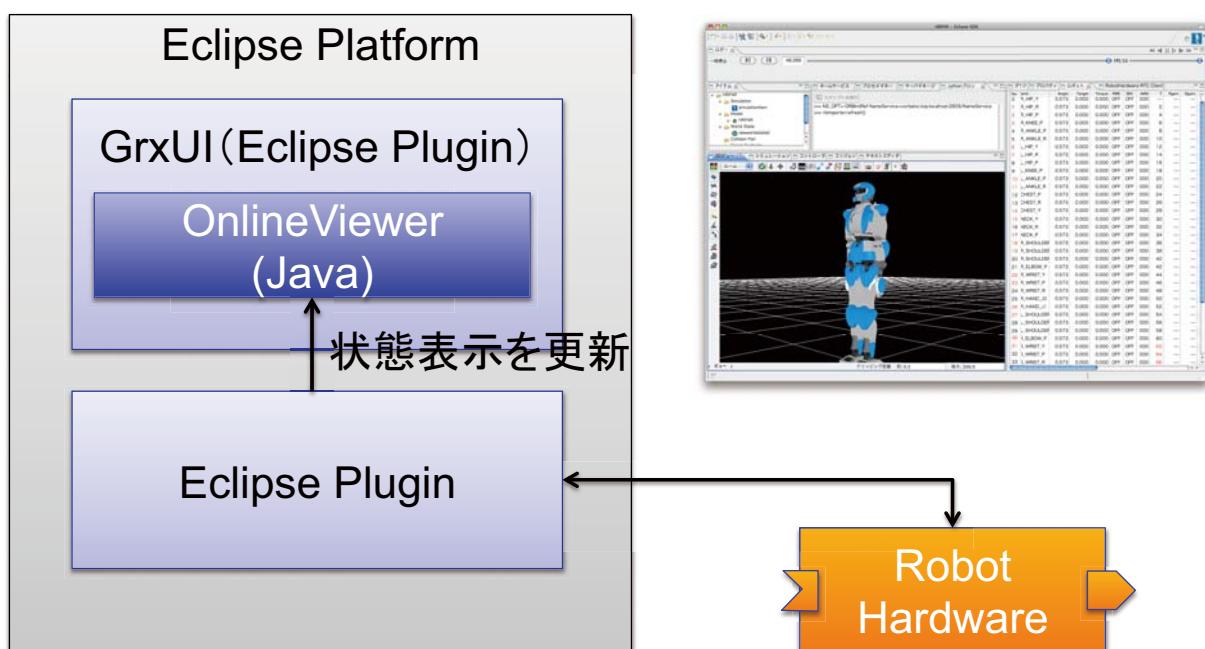
OpenHRP3/sample/example/clap/clap.cpp(CORBA経由)

OpenHRP3/sample/example/move_ankle/move_ankle.cpp(直接)

- hrpModel
 - 順運動学
 - 逆運動学
 - 順動力学
- hrpCollision
 - ポリゴン集合間の干渉の有無
 - ポリゴン集合間の最短距離
 - 光線とのポリゴン集合の干渉の有無, 距離
 - ポイントクラウドとポリゴン集合の干渉の有無

独立行政法人 産業技術総合研究所

GrxUIをロボットのモニタとして利用する



独立行政法人 産業技術総合研究所