RtcHandle - 使い方とそれを用いたRTC利用環境の構築

目次

使い方とそれを用いたRTC利用環境の構築

はじめに

OpenRTM-aistを用いてRTCを作ることができるようになってきていますが、作られたRTCを用いてロボットシステムを構築し、ロボットアプリケーションを実現することに関して は、簡単な方法はまだ提供されていません。 ここでは、RTCを利用するためのツールとしてのrtc_handle.pyを作成したのでその使い方などを説明します。最終的には、これを利用したロボットアプリケーションプログラミング環境が実現できれば良いなと考えています。

ロボットの作業プログラミング

すでにあるRTCを使うために、C++でRTCを作成して、そのonExecute関数の中からメニューを出して選択させるなどということをしていないでしょうか？

たとえば、

```
select menu(1:set param, 2:set destination, 3:move arm) >>
```

などといった感じで。できたロボットシステムをちょっと試すだけならこれでも良いかもしれません。でも、

> **場所Aにあるコップを取って、場所Cに置いて、場所Dにあるポットを取り、コップに注ぎ、ポットを場所Dに置く**

という作業をさせたいときはどうしたらよいでしょうか。このロジックをRTCのonExecuteに置くのは少し奇異な感じがしませんか？また、場所Aやコップが変わるものの場合はどうするのでしょうか。入力プロンプトを出して手入力しますか？データポートで受け取りますか？それともコンフィギュレーションサービスを使いますか？

一般にはこの部分はロボット作業プログラミングシステムの役割です。これには昔のロボット言語のようなテキストベースのものから mindstormのようなグラフィカルなものまでいろいろあります。 RTCLinkは、なんとなく似ているように見えますが、 RTCのモニタリングや簡単な操作が主であり、プログラミング環境とはまったく異なるものです。ロボットシステムの記述や構築とロボット作業の記述や実行は大きく異なる階層のものであり、それぞれ別の「モデル」を持つものです。

もちろん「作業をプリミティブ動作を実現するRTCに分解して、それを組み合わせることでアプリケーションを記述する。」という考え方もあります。この「作業をプリミティブ動作に分解してそれを組み合わせることでアプリケーションを記述する。」という考え方は基本的には正しい考え方です。しかし、そのプリミティブ動作をRTCで実現するのが良いかどうかは不明です。これはこれで、一つの（面白い）研究対象だとは思いますが。

rtc_handle.pyとは

ここまで大上段にロボット作業プログラミングを論じてきましたが、 rtc_handle.pyはそれほど大それたものではなく、そこにあるRTCを外部のプログラムから簡単に操作することを目指したお助けツールです。

rtc_handle.pyでできることを簡単にまとめると以下のようになります。

- python環境からのRTCの簡単操作
  rtc_handle.pyを使うと、RTCを作ることなく他のRTCにアクセスできます。また、pythonなのでメニューは不要です。作った関数などはその場でインタラクティブに利用できます。
- OpenRTM-aistで不足しているロボットシステム構築の支援
  RTCの接続や活性化が簡単に記述できるので、OpenRTM-aistで支援が不足しているロボットシステム構築が容易になります。
- RTCおよびロボットシステムのデバッグツール
  サービスポートやデータポートに簡単にアクセスできるので「ちょっと試す」だけなら非常に強力なツールです。これを利用してRTCやロボットシステムのデバッグを行うのも便利です。
- RTCのプロトタイピング
  これらの機能を使えば、他のRTCを利用しながら動作する新しいRTCの機能を事前に簡単に試すことができるので RTCのプロトタイピングにも利用できます。
- ロボット作業アプリケーションの開発、実行
  RTCを操作するだけでなく、pythonのプログラミング環境をそのまま使えるので容易にロボットアプリケーションを開発し、実行できるようになります。
- アプリケーションプログラミング環境の構築ベース
  最終的には、それぞれの分野でロボット作業の「モデル」を考え、それに基づいてロボット作業アプリケーションプログラミング環境を構築してもらえればよいと考えています。

前提とするOpenRTMシステム

プログラムのソースは rtc_handle.py**です。**Eclipse Public License**にしたがって配布します。（そのうちちゃんとしたライセンス条項を書かないとダメですね。）**

linux

取り合えず私の開発環境です。

- OpenRTM-aist-0.4.2
  - OpenRTM-aist-0.4.2**の未接続**port**の**disconnect**に関するパッチ(**OpenRTM-Bug**)は** rtc_handle**側で対処したので不要になりました。とおもったら未だ必要でした。**
- OpenRTM-aist-python-0.4.1
- **確認は**Vine 4.2**のみ。おそらく**Linux**ならOK。**

Windows XP

試したのは以下の環境

- Python2.5(**もともと**sourceforge**からのものを入れてあった**)
  - python-2.5.1.msi**でも大丈夫だと思います。**
- omniORBpy-3.1.msi
- OpenRTM-aist-Python2.5-0.4.1-RELEASE.msi

および

- python-2.4.4.msi
- omniORBpy-2.7.msi
- OpenRTM-aist-Python2.4-0.4.1-RELEASE.msi

少しpythonのこと

いろいろとインタラクティブに試しながらの開発するのにpython**は非常に便利です。前にも書いたように、メニューなどを書かずにその場でいろいろプログラムを書きながら試せます。**

また内観機能が強力で便利です。たとえば、aho という変数が何者かわからなかったら、

```
>>> aho
<rtc_handle.RtcOutport instance at 0xb6e7348c>
```

という感じで確かめることができます。さらに、

```
>>> dir(aho)
['__doc__', '__init__', '__module__', 'con', 'data_class', 'get_info', name',
'port_profile', 'prop', 'read', 'ref']
```

として、またさらに、

```
>>> aho.port_profile
```

```
<RTC.PortProfile instance at 0xb6e7346c>
>>> dir(aho.port_profile)
['_NP_RepositoryId', '__doc__', '__init__', '__module__', connector_profiles',
'interfaces', 'name', 'owner', 'port_ref', 'properties']
>>>
```

などとすれば、どんどん深く情報を掘り下げることができます。これは情報の構造が複雑なRTC/OpenRTMを解剖しながら試していくにはとても便利な機能です。python版のOpenRTM-aistは基本的にはC++や他の言語のものと同じですので、ここで理解したことは他の言語のOpenRTM-aistでもかなり役に立ちます。

また rtc_handle.pyを書く過程で、OpenRTM-aist自身のバグを発見したり、機会があれば別途紹介しますが、

- **pythonでRTCを走らせ、onExecuteを使わずにpython内部からRTCとして外のRTCにアクセスする、**
- **pythonでRTCを走らせ、後から内部でonExecuteなどの関数を作成する、**
- **rtc-templateを使わずに(RTMのモジュールは使う)RTCを作成する、**
- **ポートも何もない空のRTCを走らせ、後から内部でそれらを作成する、**

ということができています。

とにかく分からなくなったら dir(...) をお忘れなく。

pythonを使っている人にはいわずもがなことも多い思いますが、このあともpythonの使い方も含めて書いていきます。

rtc_handleの基本的な使い方

rtc_handleのimport

rtc_handle.pyにパスが通っている必要があります。簡単には、そこのディレクトリに行けばたいてい大丈夫です。環境変数PYTHONPATHに加える手もあります。手入力だと面倒ですが、わたしが好きなのは、pythonの中でsys.pathを一時的に変更する方法です。

```
import sys
RtmToolsDir="/..iroiro../tools"

save_path = sys.path[:]
sys.path.append(RtmToolsDir+"/rtc_handle")
from rtc_handle import *
sys.path = save_path
```

Rtm環境の立ち上げ

OpenRTM-aistは、分散オブジェクトプラットフォームとしてcorbaを使っています。そのためにはorbを立ち上げたり、COSNamingへのアクセスが必要になります。それらを引き受けているのがRtmEnvです。

一番目の引数はomniORBのオプション、これは無視して構いません。二番目の引数はネーミングサービス名のリストです。次に、ネーミングサービス名を指定して、オブジェクトのリストアップを行います。結果は、env.name_space["localhost:9876"].obj_listに辞書形式で保持されます。同時に、env.name_space["localhost:9876"].rtc_handlesに辞書形式で識別されたRTCのRtcHandleが保持されます。

```
env = RtmEnv(sys.argv, ["localhost:9876"])
env.name_space["localhost:9876"].list_obj()
```

たとえば、

```
>>> env.name_space.keys()
['localhost:9876']
```

これでlocalhost:9876というネーミングサービスがあるのが分かります。さらに以下のようにして登録されているrtcをリストアップしてrtc_handleを作ります。

```
>>> env.name_space['localhost:9876'].list_obj()
objcet localhost¥.localdomain.host_cxt/ConfigSample0.rtc was listed.
localhost¥.localdomain.host_cxt/ConfigSample0.rtc is not alive.
objcet EmptyRtc0_py.rtc was listed.
EmptyRtc0_py.rtc is not alive.
objcet move0.rtc was listed.
handle for move0.rtc was created.
objcet mixer0.rtc was listed.
handle for mixer0.rtc was created.
objcet frm_ctrl0.rtc was listed.
handle for frm_ctrl0.rtc was created.
objcet pa10fk0.rtc was listed.
handle for pa10fk0.rtc was created.
objcet jinv0.rtc was listed.
handle for jinv0.rtc was created.
objcet vel_7dof0.rtc was listed.
vel_7dof0.rtc is not alive.
objcet pa10disp0.rtc was listed.
handle for pa10disp0.rtc was created.
[['localhost¥¥.localdomain.host_cxt/ConfigSample0.rtc', <RTC._objref_DataFlowCom
ponent instance at 0xb6e6d0ec>], ['EmptyRtc0_py.rtc', <RTC._objref_DataFlowCompo
nent instance at 0xb72d1e8c>], ['move0.rtc', <RTC._objref_DataFlowComponent inst
ance at 0xb6e69dac>], ['mixer0.rtc', <RTC._objref_DataFlowComponent instance at
0xb6e6d1ec>], ['frm_ctrl0.rtc', <RTC._objref_DataFlowComponent instance at 0xb6e
71c2c>], ['pa10fk0.rtc', <RTC._objref_DataFlowComponent instance at 0xb6dff20c>]
, ['jinv0.rtc', <RTC._objref_DataFlowComponent instance at 0xb6e08ecc>], ['vel_7
dof0.rtc', <RTC._objref_DataFlowComponent instance at 0xb6e11fcc>], ['pa10disp0.
rtc', <RTC._objref_DataFlowComponent instance at 0xb6e15b8c>]]
```

という感じでメッセージが出ます。naming serviceに登録されているものは とりあえずcorba objectとしてリストアップされます。オブジェクトリファレンスをrtcとしてnarrowできれば、rtc_handleの作成をしま

す。作成に失敗すれば、そのrtcは生きていないと判断しています。

これで準備完了。あとは好きに使うということになります。

利用するRtcHandleの変数への割り付け

以下のようにすると、どんな名前のrtcがあるか再確認できます。

```
>>> env.name_space['localhost:9876'].rtc_handles.keys()
['pa10disp0.rtc', 'jinv0.rtc', 'vel_7dof0.rtc', 'move0.rtc', 'mixer0.rtc',
'pa10 fk0.rtc', 'frm_ctrl0.rtc']
```

しかし、たとえば pa10fk0.rtc という名前のRTCにアクセスするのにいちいち、

```
env.name_space["localhost:9876"].rtc_handles["pa10fk0.rtc"]
```

とするのは、とても面倒なので必要なRTCを変数にいれておきましょう。

```
handle1 = env.name_space["localhost:9876"].rtc_handles["pa10fk0.rtc"]
handle2 = env.name_space["localhost:9876"].rtc_handles["frm_ctrl0.rtc"]
handle3 = env.name_space["localhost:9876"].rtc_handles["vel_7dof0.rtc"]
```

ここで、

```
>>> dir(handle1)
['__doc__', '__init__', '__module__', 'activate', 'configuration_ref',
'deactivate', 'env', 'execution_contexts', 'inports', 'name', 'outports',
'port_refs', 'ports', 'profile', 'prop', 'retrieve_info', 'rtc_ref', 'services']
```

とすると、handle1がどんな属性を持っているかわかります。さらに、

```
>>> handle1.outports
{'jacob': <rtc_handle.RtcOutport instance at 0xb6ecf64c>, 'frame':
<rtc_handle.RtcOutport instance at 0xb6ecaf8c>}
```

などとすると、frameという名前の出力ポートを持っていることが分かります。

外部RTC間のポートの接続

このhandle1の出力ポートframeとhandle2の入力ポートref_frmを接続するには以下のようにIOConnectorを生成して、それに対してconnect命令を出します。

```
con1 = IOConnector([handle1.outports["frame"], handle2.inports["ref_frm"]])
con1.connect()
con1.disconnect()
```

サービスポート同士の接続の場合もServiceConnectorを使うだけであとは同じです。

```
service_con1 = ServiceConnector([handle_local.services['Con1'],handle_drive.services['
```

```
service_con1.connect()
```

注意する点は、ポート単位で接続するということ。直接corbaのserverとclientを接続してはいけません。

ポートの接続状態のチェック

各ポートでget_connnections()を発行することで、そのポートの接続情報としてConnectorProfileのリストを取得することができます。具体的には、

```
>>> handle1.outports["frame"].get_connections()
[<RTC.ConnectorProfile instance at 0xb6ec840c>]
```

という感じになります。

接続属性の変更(ConnectorProfileのproperties)

subscription_typeなどの接続属性はConnectorProfileのpropertiesに NVlistの形式で保持されています。それを変更するためには、 Connectorのprop_dictを変更して、各Portの属性との調整をしてpropertiesに保存することになります。 具体的には生成時でしたら

```
con1 = IOConnector([handle1.outports["frame"], handle2.inports["ref_frm"]], prop_dict=
```

などとしますし、生成後でしたら、

```
con1.prop_dict['dataport.subscription_type']='Periodic'
con1.nego_prop()
```

などとすることで変更できます。

ポートへの直接アクセス

RTCの各ポートは、ポート情報のやり取りやポートの接続を行うもので、実際のコマンドやデータの通信は行いません。一見不思議で煩わしいように見えますが、RTCの制御と実際のRTC同士の通信を分けることで、たとえばデータ通信に限って高速なネットワークを利用できるような枠組になっています。rtc_handleの場合、OpenRTM-aistのcorbaで実装されたデータ通信オブジェクトを前提にして、その通信オブジェクトに直接アクセスできるようにしてあります。

handle1の出力ポートframeから直接データを読むには以下のようにします。(現在この機能はJAVA版のRTCに対しては使えません。)

```
>>> handle1.outports["frame"].read()
<RTC.TimedFloatSeq instance at 0xb72fb6ec>
```

戻ってくるのは時間付のデータになります。一方、入力ポートに直接データを送るときは、以下のように時間なしのデータでwriteを呼び出せば時間部を付けて送ります。ただ、今は単に RTC.Time(0,0)を付けているだけなので、今後、ちゃんとunixのエポック時間を付けるか、ユーザに任せるか、変更が入る可能性があります。

```
>>> handle2.inports["ref_frm"].write(data)
```

data型は以下のようにすると分かります。

```
>>> handle2.inports["ref_frm"].prop["dataport.data_type"]
'TimedFloatSeq'
```

OpenRTMの標準のDouble,FloatとかIntならpythonの実数、整数を、また XxxSeq ならそれらのリストで渡してやればOKです。

サービスポートに関しては、ユーザクラスのnarrowingやデータ型の問題があるので次項で説明します。ためしに

```
>>> pa10fk.services['ComFk'].provided['com_fk'].ref
```

としてみて下さい。これがサービスポートのサーバのオブジェクトリファレンスなのですが、

```
<omniORB.CORBA.Object instance at 0xb6e9216c>
```

となっているはずです。つまり一般のCORBA.Objectとして扱われているのでユーザが定義したoperationは、まだ使えない状態になっています。

RTC利用環境の構築例

RTC利用環境の構築には、必要なRTCの起動やRTCの事前接続、活性化などが含まれます。また、ユーザ定義のサービスポートやデータに対応したモジュールの読み込みが必要になります。さらには RTCとは直接間系がなくても必要なpythonモジュールを読み込むこともあります。rtc_handleはその一部の支援を行っているに過ぎません。以下にわたしが作ったpa10アームの分解運動速度制御を実現するRTC環境の立ち上げの例を示します。

事前の利用準備

必要なサービスのidlのコンパイル。

RTCのサービスポートを使うためには、それに対応したidlファイルが必要になります。以下のようにomniORBでidlコンパイルを行うと、 module宣言があれば、そのモジュール名 (たとえば Mmm)を用いて MmmおよびMmm__POAというでディレクトリが作られます。

```
$ omniidl -bpython xxx.idl
```

スタブはMmm、スケルトンはMmm__POAをモジュールとして読み込むことになります。サービスポートのprovidedポートをpythonからclientとして利用するには Mmmをimportすれば良いということです。サービスポートのrequiredにpythonからserverとして接続するには Mmm__POAを読み込み、実装コードを書く必要があります。しかし、serverとして接続するにはportとして接続して相手側にserverの存在を教える必要がありますが、現在のrtc_handleでは、pythonと外部ポートとをポート接続することをサポートしていないのでここはとりあえずパスします。

現状のOpenRTMのrtc-templateはきちんとmoduleをサポートしていません。rtc-templateを使わずにRTCを作ればmoduleを使うことができますが、当面はmodule宣言なしでidlを作成することをお薦めします。その場合、作られるディレクトリは、_GlobalIDLおよび_GlobalIDL_POAになります。これらの実体は、その下に_init_.pyがあって、上にあるコンパイル結果 xxx_idl.pyを読み込んでいるだけです。

pythonの場合、同じ名前のモジュールは1つでなくてはならないので使うサービスのスタブは同じ_GlobalIDLに入れる必要があります。_init_.pyはomniidlをやるたびに逐次追記されるので、必要なものを同じディレクトリで1つずつidlコンパイルしていけば良いことになります。

必要な実装のプログラム作成。

上に述べたようにサービスポートのrequiredにpythonからserverとして接続することは今のところrtc_handleではサポートしていないので割愛します。

RTC利用環境の記述

RTC利用環境を簡単に行うには、事前に環境の記述があってそれを読み込んでなるべく自動的に構築される必要があります。記述したい内容は、たとえば次のようなものがあります。

- NamingService
  - **複数を可能とする**
- **利用**RTC
  - NamingService、name_string
  - RtcHandle**をバインドする変数名**
  - service port**の**proxy**ファイル**
- **接続記述**
  - **種別：データ、サービス**
  - **接続ポート：**handle**変数**.inport[**ポート名**]、、、、
  - connector**変数名（自動生成？）**
- connection
  - **事前に接続する**connector**とその順番**
- activation
  - activate**する**RTC**とその順番**

一般には、このような記述は実装に中立な、たとえばxmlなどで記述するのが良いとされています。しかし、やろうとしてみると分かるのですがxmlの記述は決して人間にとって読み書きしやすいものではないし、複雑なことを記述しようとするととてつもなく繁雑で長くなります。また、環境構築でやりたいことの中には実装依存の部分もかなりあります。それを無理矢理に実装非依存な部分と実装依存の部分に別けて、それらをマージしながら解釈していくのは大変な無駄と労力が必要になります。

ここではあきらめて思いっきり実装依存、pythonスクリプトで記述することにしました。後ろの項で詳しく説明します。

実装非依存な記述については別な研究開発項目として考えていただければと思います。わたしの予想としては、アプリケーション分野や利用形態をしぼっていけば、そのような記述も可能だろうと思っています。現状では、まだその途中段階です。

必要なRTCの起動（その1）

ここはハードウェアやpythonスクリプトから起動できないRTCなどを事前に 立ち上げておくという話です。事前の準備というより直前の作業かな。 後で述べるスクリプトでは、RTCへのアクセスを発生させる前に必要なRTCを立ち上げたかどうかの確認のプロンプトを出すように してあります。そういうことをxmlで記述するのは難しいと思います。

RTC利用環境構築スクリプト

必要なモジュールの読み込み

まず、必要なモジュールを読み込みます。 sysは、sys.argv、timeは、time.sleep()、osは、os.popen2()に必要です。 後はrtc_handle。 その他、わたしの場合、自前の幾何演算モジュールgeoをよく使います。

```
RtmToolsDir="/home/my_hom/openrtm/tools"
MyRtcDir="/home/my_home/openrtm/myRTC"

import sys
import time
import os

save_path = sys.path[:]
sys.path.append(RtmToolsDir+'/rtc_handle')
from rtc_handle import *
sys.path.append(MyRtcDir+'/tools/geo')
from geo import *
sys.path = save_path
```

スタブのimport

必要なスタブを読み込みます。以下はカレントディレクトリにある(または事前にパスを通してある)場合です。 _GlobalIDLしか使えない場合は、この仮定はそれほど不自然ではないと思います。 module宣言が使えるなら、どこか一ヶ所にまとめるというのも考えられます。

```
#
# import stub files
#
import _GlobalIDL
```

必要なRTCの起動の関数

この後は、類似の作業単位で関数にまとめて使い易くしています。 べた書きにしても良いのですが、読み込んだ後、手作業で順に 環境構築する際には分けておいた方がやりやすい。 RTCの起動は、OpenRTMとは直接関係なくosモジュールのop.popen2を使っています。 pythonライブラリリファレンスによれば、unixでもwindowsでも使えるとのこと。もちろんコマンドは異なりますが。 ここでは、各RTCがあるディレクトリに移動して、xtermを開いてそこで 走らせています。 この形でRTCを起動しておくと、pythonプロンプトでctrl-Cを打つと 全部終了させることができます。逆に、先にctrl-Dでpythonを抜けると 一つずつ終了させなくてはならなくなるので要注意です。

```
#
# run components
#
def run_components() :
```

```
com_mixer="cd "+MyRtcDir+"/mixer_cb; xterm -e ./mixerComp"
com_jinv="cd "+MyRtcDir+"/jinv_cb; xterm -e ./jinvComp"
com_frm_ctrl="cd "+MyRtcDir+"/frm_ctrl_cb; xterm -e ./frm_ctrlComp"
com_pa10fk="cd "+MyRtcDir+"/pa10fk_cb; xterm -e ./pa10fkComp"
com_vel_7dof="cd "+MyRtcDir+"/vel_sim_df_2port; xterm -e ./vel_7dofComp"
os.popen2(com_mixer)
os.popen2(com_jinv)
os.popen2(com_frm_ctrl)
os.popen2(com_pa10fk)
os.popen2(com_vel_7dof)
com_pa10disp="cd "+MyRtcDir+"/pa10disp_cb; xterm -e python pa10disp.py"
os.popen2(com_pa10disp)
```

必要なRtcHandleの変数への割り付けの関数

前述のように、RtcHandleへのアクセスを簡単にするために変数に割り付けます。 その変数はglobal宣言をしておくことでmoduleの外部からアクセスできるように なります。スクリプトの読み込みを、

```
import モジュール
```

とした場合は、モジュール.変数名、

```
from  モジュール import *
```

とした場合は、変数名で直接アクセスできます。

ここでは同時にサービスポートのserverのnarrowingやその引数などに使う ユーザ定義データ型の処理を行っています。異質なものが混ざっていて汚い と言う御批判もあると思いますが、とりあえず御容赦を。

で、rtc_handleのnarrowingでは、クラスを表す文字列をevalしています。 そのとき正しくevalするためには、_GlobalIDLなどが見えなくてはいけないのですが 普通ではrtc_handleモジュールからは見えません。そこで環境辞書を渡しています。 そのときのグローバル環境が globals()になります。 ユーザ定義データは、データ型をc_frameに入れて後で利用できるようにしています。 m_tool_frmにインスタンスを入れていますが、 モジュールを読み込んだ後、_GlobalIDLが見えなくなっても これと同じ形式で使えます。

引数envはRtmEnvです。RtmEnvは後でグローバル変数に入れるし、 関数と言うよりスクリプトのブロックでしかないのでの、 引数にする必要もないのですがなんとなく引数にしています。

```
def assign_variables(env) :
 # rtc
 global vel_7dof, pa10fk, frm_ctrl, mixer, jinv, pa10disp, move
 vel_7dof = env.name_space["localhost:9876"].rtc_handles['vel_7dof0.rtc']
 pa10fk = env.name_space["localhost:9876"].rtc_handles['pa10fk0.rtc']
 frm_ctrl = env.name_space["localhost:9876"].rtc_handles['frm_ctrl0.rtc']
 mixer = env.name_space["localhost:9876"].rtc_handles['mixer0.rtc']
```

```
jinv =env.name_space["localhost:9876"].rtc_handles['jinv0.rtc']
pa10disp = env.name_space["localhost:9876"].rtc_handles['pa10disp0.rtc']
move = env.name_space["localhost:9876"].rtc_handles['move0.rtc']
# narrow services
pa10fk.services['ComFk'].provided['com_fk'].narrow_ref(globals())
# data & class
global c_frame, m_tool_frm
c_frame=_GlobalIDL.Frame
m_tool_frm = c_frame(MATRIX().val,VECTOR().val)
```

利用するコネクタの生成の関数

利用するコネクタを事前に準備しています。準備したコネクタはenv.con_listにリスト形式で保持します。オブジェクト指向のカプセル化には大きく外れますが、こういうことができるのがpythonの便利なところです。もともとRtmEnvやRtcHandleはオブジェクトというより データホルダーや構造体としての役割が大きいのです。なぜ辞書ではなくリストかという理由は、コネクタを作って以下のようにしてみると分かります。

```
>>> make_connectors(env)
>>> for con in env.con_list :
...    print con.name
...
th_th
th2_th
frame_cur_frm
jacob_jacob
frame_ref_frm
state_state
vel_vel
j_list_y_jacob
vel_vel
```

そう、同じ名前のものがあります。これはコネクタ名を自動生成したためで 明示的に違う名前をあたえることもできます。でもそうする手間をかけるなら違う名前の変数に割り当てた方が 使い易いと思います。

```
def make_connectors(env):
 env.con_list=[]
 con = IOConnector([vel_7dof.outports['th'],pa10fk.inports['th']])
 env.con_list.append(con)
 con = IOConnector([vel_7dof.outports['th2'],pa10disp.inports['th']])
 env.con_list.append(con)
 con = IOConnector([pa10fk.outports['frame'],frm_ctrl.inports['cur_frm']])
 env.con_list.append(con)
 con = IOConnector([pa10fk.outports['jacob'],mixer.inports['jacob']])
 env.con_list.append(con)
 con = IOConnector([move.outports['frame'],frm_ctrl.inports['ref_frm']])
 env.con_list.append(con)
 con = IOConnector([frm_ctrl.outports['state'],move.inports['state']])
```

```
 env.con_list.append(con)
 con = IOConnector([frm_ctrl.outports['vel'],mixer.inports['vel']])
 env.con_list.append(con)
 con = IOConnector([mixer.outports['j_list'],jinv.inports['y_jacob']])
 env.con_list.append(con)
 con = IOConnector([jinv.outports['vel'],vel_7dof.inports['vel']])
 env.con_list.append(con)
```

コネクタの接続の関数

とりあえずenv.con_listにあるコネクタを全部接続しています。

```
def connect_components(env) :
 for con in env.con_list :
   con.connect()
```

RTCの起動の関数

RTCを接続します。RtcHandleには変数名でアクセスしているので envは関係なしなので引数にもしていません。 ここでは起動の順序や待ち時間の調整もしています。

```
def activate_components() :
 vel_7dof.activate()
 time.sleep(1)
 pa10disp.activate()
 pa10fk.activate()
 move.activate()
 time.sleep(5)
 frm_ctrl.activate()
 time.sleep(5)
 mixer.activate()
 time.sleep(3)
 jinv.activate()
```

最後に実行文

環境を作って、RTCを起動して、他のRTCの起動を確認して、 RtcHandleを生成して、変数に割り当てる、ところまでをやっています。 コンポーネントの接続や活性化はモジュールを読み込んだ後、 手作業でやるのうにしました。もちろんここで全部やっても良いのですが 様子見段階ではこういうこともできるということです。

```
env = RtmEnv(sys.argv,["localhost:9876"])
run_components()
time.sleep(10)
raw_input("are you sure that every rtc needed are running")
for ns in env.name_space :
 env.name_space[ns].list_obj()
```

```
assign_variables(env)
```

実行例

前述のスクリプトの実行例です。

pythonの起動とスクリプトモジュールの読み込み

pa10_testがモジュール名です。読み込むとxtermが開いてそれぞれRTCが起動されます。しばらくすると、必要なRTCが起動されているか聞いてくるので、リターンすると(ここではyesとかnoの判断はしていません) オブジェクトのリストアップとRtcHandleの作成を行います。

このときにリストアップしたRTCが死ぬようならばOpenRTMにパッチを あてる必要があります。

```
$ python
Python 2.4.4 (#1, Apr 25 2008, 09:54:36)
[GCC 3.3.6 release (Vine Linux 3.3.6-0vl7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pa10_test import *
are you sure that every rtc needed are running?
objcet move0.rtc was listed.
handle for move0.rtc was created.
objcet mixer0.rtc was listed.
handle for mixer0.rtc was created.
objcet jinv0.rtc was listed.
handle for jinv0.rtc was created.
objcet frm_ctrl0.rtc was listed.
handle for frm_ctrl0.rtc was created.
objcet pa10fk0.rtc was listed.
handle for pa10fk0.rtc was created.
objcet vel_7dof0.rtc was listed.
handle for vel_7dof0.rtc was created.
objcet pa10disp0.rtc was listed.
handle for pa10disp0.rtc was created.
```

RTCの接続と活性化

コネクタの生成、接続、RTCの活性化のルーチンを呼びます。これで、分解運動速度制御のRTC群が使えるようになります。pa10の手先座標系での目標指令値を生成するmove0.rtc(python版RTC)の コンソールから、

```
>>> goto(comp,pos_a)
```

などとやると、pa10アーム(ここでは幾何モデル、実機も同じ)が動きます。

```
>>> make_connectors(env)
>>> connect_components(env)
>>> activate_components()
```

サービスポートの利用例

pa10の順運動学計算をやるpa10fk.rtcは、ツール座標系を指定する サービスポートを持っています。そのオブジェクトレファレンスは、pa10fk.services['ComFk'].provided['com_fk']となっています。 そこにset_toolというオペレーションが定義されています。

set_toolの引数は、corbaのidlで定義された座標系や座標変換をあらわす Frame型（この環境では _GlobalIDL.Frameクラス）です。 ここには他にわたしがよく使う幾何演算用のgeo.FRAMEクラス（演算なども定義されている）もありますし、後で出てくるoutportのFloatSeqで表現されたframeもあります。 それぞれ座標系を表現するものですが、これらをどのように扱っていくのが良いかは 今後の課題でしょう。

```
>>> pa10fk.services['ComFk'].provided['com_fk']
<rtc_handle.CorbaServer instance at 0xb6eb7dcc>
>>> aho=pa10fk.services['ComFk'].provided['com_fk']
>>> aho.ref
<_GlobalIDL._objref_ComFk instance at 0xb6eb7f0c>
>>> aa=FRAME(xyzabc=[100,200,0,pi/2,0,0])
>>> m_tool_frm.mat
[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
>>> m_tool_frm.mat=aa.mat.val
>>> m_tool_frm.vec=aa.vec.val
>>> m_tool_frm.mat
[[1.0, 0.0, 0.0], [0.0, 6.1230317691118863e-17, -1.0], [0.0, 1.0,
6.1230317691118863e-17]]
>>> m_tool_frm.vec
[100.0, 200.0, 0.0]
>>> aho.ref.set_tool(m_tool_frm)
```

出力ポートからのデータの取得

pa10fkのframeという名前のoutportからデータを読んでみましょう。 時間つきの実数シーケンスになっています。そのデータ部を見ると、 空リストになっています。これは出力ポートに複数の入力ポートを 接続できないというOpenRTM-aist-0.4.2のバグです。

```
>>> baka=pa10fk.outports['frame']
>>> baka.read()
<RTC.TimedFloatSeq instance at 0xb6ece92c>
>>> baka.read().data
[]
```

そこでこの出力ポートの接続をいったん切ります。確認のためコネクタ名を リストアップします。3番目のframe_cur_frmがそれなのでdisconnectを行います。 切断後はちゃんとデータが読めます。

```
>>> for con in env.con_list :
...    print con.name
...
th_th
```

```
th2_th
frame_cur_frm
jacob_jacob
frame_ref_frm
state_state
vel_vel
j_list_y_jacob
vel_vel
>>> env.con_list[2].disconnect()
>>> baka.read().data
[-0.70709842443466187, 0.7071150541305542, 1.1607184205786325e-05,
0.70711511373519897, 0.70709848403930664, -1.3943366866442375e-05,
-1.8061688024317846e-05, -1.6360875179088907e-06, -1.0000001192092896,
599.9996337890625, 299.9178466796875, 300.0545654296875]
```

終了

ctrl-Cでpythonから起動したRTC(のxterm)を全部終了させます。その後、ctrl-Dでpythonから抜けます。

```
>>>
KeyboardInterrupt
>>>
$
```

configuration service

内部構造

- rtc_templateでの定義

```
--config=int_param0:int:0
--config=パラメタ名:パラメタ型:default値
```

この値は、'default'というconfiguration setとしてコンポーネントの初期化時にセットされます。

- rtc本体での初期化
  変数宣言として__init__で以下のようになっています。

```
self._int_param0=[0]
self._パラメタ名=[default値]
```

ユーザがアクセスする変数領域。値をリストに入れているのはpythonが基本的に call_by_valueであるため、関数などにself._int_param0を引き渡して値をもらって戻ってくるには、このような構造体にして受け渡す必要があります。

- configuration setと変数領域のバインディング

```
self.bindParameter("int_param0",self._int_param0, "0")
self.bindParameter("パラメタ名",self._パラメタ名, "default値")
```

これにより'default'のconfiguration setに"パラメタ名"、"default値"のNamedValueがセットされます。valueはstringのanyになることに注意してください。同時に、ユーザがアクセスする変数領域もバインドされます。configuration serviceにactivate_configurationを発行したときに configuration setの値が変数に代入されます。外部からの話になりますが、configuration setの値を変更した後、それを rtcの振舞に反映させたければ、以下を行う必要があるということです。

```
conf_ref.activate_configuration('default')
conf_ref.activate_configuration('コンフィグ名')
```

configuration setの切替時だけでなく、そのときにactiveなconfiguration setのパラメタを変更した場合にも このオペレーションを発行しないとユーザが使う変数へ反映されません。

rtc_handleでの使い方

handle.conf_ref

configuration serviceのオブジェクトリファレンス

handle.conf_set

configuration setの辞書。default setは、handle.conf_set['default']で出てきます。

handle.conf_set_data

configuration setのデータ辞書。configuration setではデータはNVListで保持されているのでpython辞書形式に変換しています。

たとえばデフォールトのconfiguration setのparam1の値を見たければ、

```
handle.conf_set_data['default']['param1']
```

とします。その値を変更したければ、

```
handle.set_conf('default','param1','1')
```

などとします。変更した値を反映させたければ、

```
handle.conf_ref.activate_configuration_set('default')
```

とします。変更時にすぐに反映させたければ、

```
handle.set_conf_activate('default','param1','1')
```

とします。

アプリケーションの作成へ向けて

前項のようなスクリプトを書けば、そこにあるRTCを扱うための自分なりの 環境が構築できます。 この上でどのようなアプリケーションを開発するかは、各々の課題です。 分散型ロボット化環境を構築してイベントドリブンでRTCを連携させる、 RTCLinkのようなGUIでのアプリケーションプログラム環境を構築する、などなど、いろいろ考えられます。それぞれのアプリケーションで RTC利用環境構築は異なるものになるでしょう。イベントドリブンや GUIプログラミングでは、RtcHandleを事前に静的に変数に割り当てる 必要はないかもしれません。

いずれにしてもアプリケーションが標準的なRTCを使うように 書かれていれば、RTCではなく、そのアプリケーションの再利用性が向上します。 それが体系化されていけば、RTCの上位のアプリケーションフレームワークが できあがることになります。rtc_handleは、RTCとそのような新しいアプリケーション フレームワークとの橋渡しになればと思っています。

残っている課題

出力ポートからpush型でデータを受け取るにはどうしたら良いか。 サービスポートのclientに接続するにはどうしたら良いか。 これらは利用環境内部でのポートの生成が必要になり、利用環境のRTC化に かかわる問題になります。

これについては、empty_rtcという、空のRTCを走らせて後から サービスや入出力のポートを追加するモジュールを作成中なので まとまればまた紹介します。

rtc_handleに関するいくつかの議論

この項目は、検討過程のものをそのままのせていて未整理状態です。

- RTCの起動
  利用環境と独立な場合もあるし、利用環境から起動したい場合もある。 起動方式を指定することも考えられるがここでは棚上げ。
- RTCのServiceポートのプロキシ
  IDL定義からそのときに生成する方法もある（要検討）。 とりあえずは、omniPyでコンパイル済みのスタブファイルがあるとする。 ファイル（やリポジトリなどのURI）を指定して読み込む。
- RTCのServiceポートの実装
  接続したいServiceポートがProviderを持っている場合はそれを実装する必要がある。 そんな場合があるか？ あるとして実装はどうやって読み込む？
- 利用環境のRTC化
  外部RTCのポートへ接続をポートを使ってやるならRTC化するのが良い。 直接CORBAオブジェクトとして叩くならRTC化は不要。 OutPortからpush型で受けるならこちら側もポートの方が良い。 すなわちRTC化するということ。 両方作ってみる？
- rtc_handleについて
  RTC化した利用環境用とそうでないのと2つ作る？ それとも1つで、それぞれに付け加える？ それぞれ継承させる？
- RTC化のメリット、デメリット
  - ○接続は楽？
  - ×接続しなくては使えない。
  - ○バッファ、コールバックが使える。
  - ○データ変換、ポート状態を見れる。

    - ×直に使えない。
    - ×利用側だけならProviderはいらないことが多い(常にというわけではないが)。
    - ○逆にConsumerとProviderがセットの場合は ポートとして接続するのが楽。
      というか、外部からProviderへどうアクセスさせるかは、サービスポートとして接続するしか指定できない。つまりRTC化せずにProviderを用意するのは無意味。
- 利用環境記述はどうするか
    - 記述形式、オリジナルdata形式、xml、pythonプログラム
    - 項目
    - 接続は含めるか
- そもそも利用とは何か
    - rtcの情報を得る
    - rtcのactivateなど、状態遷移
    - ポートの接続
    - ポートへのアクセス
        - 入力ポート: データを送りつける
        - 出力ポート: データを読み込む
        - サービスプロバイダポート: サービスをリクエストする
        - サービスコンシューマポート: サービスリクエストに答える
- スタブなど(omniidl -bpython "file")
    - idlコンパイルしたディレクトリの下にモジュール名のディレクトリができる。
    - その下の"__init__.py"で、上にあるスタブファイルを読み込むようになっている。
    - idlファイルごとに逐次idlコンパイルしてもモジュール名のディレクトリの"__init__.py"に逐次追加される。
    - モジュール名無し(グローバルの場合)、"_GlobalIDL"、"GlobalIDL__POA"ができる。
    - pythonでの読み込みは、モジュール単位。したがって同じモジュール(特にグローバル)の場合、pathを変更しつつ読み込むのではだめで、一つのファイルにまとめられていないといけない。そのためには、idlファイルへのpathをもらって一箇所でidlコンパイルするというのが良い。

rtc_handle.pyのコード解説

まずは標準的なおまじない。

```
#/usr/bin/env python
# -*- Python -*-

import sys
from omniORB import CORBA, URI
from omniORB import any

import OpenRTM
import RTC

from CorbaNaming import *
```

```
import SDOPackage
```

RtmEnvのクラス定義

RtmEnvでは、プロセス全体で使うorbを生成してするとともに、RTシステム全体のいろいろな情報を収集、保持しています。orb_argsはorbを生成するときのオプションです。詳しくはomniORBのマニュアルを見る必要があります。pythonを起動するときのオプションとしてsys.argvに引きわたすこともできますし、プログラム内部で生成することもできます。nserver_namesは、ネーミングサービスのサーバ名のリストです。複数を指定して、それぞれ別々の名前空間として辞書に保持されます。既定値としてlocalhostを指定しています。この場合、localhost:2809がサーバになります。orb、namingともに、他で生成されたものを使うときのオプショナルな引数です。これらが指定されると優先的にそれが使われます。この場合、ネーミングサービスは1つだけで、"default"というに入れられます。

```
# class RtmEnv :
#       rtm environment manager
#       orb, naming service, rtc proxy list
#
class RtmEnv :
   def __init__(self, orb_args, nserver_names=["localhost"],
                                      orb=None, naming=None):
       if not orb :
           orb = CORBA.ORB_init(orb_args)
       self.orb = orb
       self.name_space = {}
       if naming : # naming can specify only one naming service
           self.name_space['default']=NameSpace(orb, naming=naming)
       else :
           for ns in nserver_names :
             self.name_space[ns]=NameSpace(orb, server_name=ns)
   def __del__(self):
       self.orb.shutdown(wait_for_completion=CORBA.FALSE)
       self.orb.destroy()
```

Name Spaceのクラス定義

ネーミングサービスに対応したNameSpaceクラスは、生成時にはCorbaNamingを作るだけです。登録オブジェクト情報をリストしたり、rtcに対してRtcHandleを生成するのは別途行うようになっています。server_nameにネーミングサーバ名を指定します。名前の解決にはorbが必要となります。namingに既に解決されたサーバが指定されればそちらを優先します。解決されたネーミングさービスはself.namingに保持されます。self.b_lenは登録されたオブジェクトをリストアップするとき一回に情報を取得する数の最大数です。とりあえず10にしてあります。self.rtc_handlesは、rtcのRtcHandleを保持するための辞書、self.obj_listは、オブジェクトリファレンスを保持するための辞書です。

```
#
# class NameSpace :
#       rtc_handles and object list in naming service
#
class NameSpace :
```

```
    def __init__(self, orb, server_name=None, naming=None):
        self.orb = orb
        self.name = server_name
        if naming :
          self.naming = naming
        else :
          self.naming = CorbaNaming(self.orb, server_name)

        self.b_len = 10 # iteration cut off no.
        self.rtc_handles = {}
        self.obj_list = {}
```

get_object_by_nameでは、指定された名前をネーミングサービスから検索して cIで指定されたクラスにnarrowしてオブジェクトリファレンスを返します。ここではcIのデフォールトはRTCになっています。強制的にNoneにしておくと、一般的なオブジェクトレファレンスになります。 pythonの場合、事前にスタブが読み込まれていれば可能であれば自動的に対応するクラスにnarrowされるようです。

```
    def get_object_by_name(self, name, cl=RTC.RTObject):
        ref = self.naming.resolveStr(name)
        if ref is None: return None     # return CORBA.nil
        if cl :
            return ref._narrow(cl)
        else :
            return ref
```

list_objでは、ネーミングサービスに登録されているオブジェクトを検索して登録名をキーにした辞書self.obj_listにオブジェクトリファレンスを入れます。オブジェクトがrtcの場合、同時にRtcHandleを生成し、別な辞書self.rtc_handlesにも入れます。冗長ですが、名前とRtcHandleのリストのリストを戻り値として返します。 list_obj1では、そのネーミングコンテクストの階層のオブジェクトをリストアップして下のコンテキストのリストアップを再帰的に呼び出すようになっています。 name_contextが指定されていない場合、デフォールトで_rootContextを使うようになっているので、self.rtc_handlesやself.obj_listの初期化を除けばトップレベルで使ってもうまくいくはずです。このへんは少しコードが冗長かな。 b_listの0番目の要素にself.b_lenを最大数としたオブジェクトリファレンスが入ります。残りがある場合は、1番目の要素がiteratorになっているので要素がなくなるまで繰り返し呼び出します。iteratorの戻り値は、0番目がオジェクトの有無、1番目の要素がオブジェクトのリストになっています。

```
    def list_obj(self) :
        self.rtc_handes = {}
        self.obj_list = {}
        return self.list_obj1(self.naming._rootContext, "")

    def list_obj1(self, name_context, parent) :
        if not name_context :
            name_context = self.naming._rootContext
        rslt = []
        b_list = name_context.list(self.b_len)
        for bd in b_list[0] :
            rslt = rslt + self.proc_bd(bd, name_context, parent)
```

```
        if b_list[1] :  # iterator : there exists remaining.
            t_list = b_list[1].next_n(self.b_len)
            while t_list[0] :
                for bd in t_list[1] :
                    rslt = rslt + self.proc_bd(bd, name_context, parent)
                t_list = b_list[1].next_n(self.b_len)
        return rslt
```

proc_bdでは、オブジェクトがネーミングコンテクストなら再帰的に掘り下げ、そうでない場合は名前とオブジェクトリファレンスのバインディングを行い辞書に登録しています。またRTC.RTObjectでnarrowして、成功すればRtcHandleを生成して辞書に登録します。

```
    def proc_bd(self, bd, name_context, parent) :
        rslt = []
        pre = ""
        if parent :
            pre = URI.nameToString(parent) + "/"
        nam = pre + URI.nameToString(bd.binding_name)
        if bd.binding_type == CosNaming.nobject :
            tmp = name_context.resolve(bd.binding_name)
            self.obj_list[nam]=tmp
            print 'objcet '+nam+' was listed.'
            try :
                tmp = tmp._narrow(RTC.RTObject)
            except :
                print nam+' is not RTC.'
                tmp = None
            try :
                if tmp :
                   rslt = [[nam, tmp]]
                   self.rtc_handles[nam]=RtcHandle(nam,self,tmp)
                   print 'handle for '+nam+' was created.'
                else :
                   pass
            except :
                print nam+' is not alive.'
                pass
        else :
            tmp = name_context.resolve(bd.binding_name)
            tmp = tmp._narrow(CosNaming.NamingContext)
            rslt = self.list_obj1(tmp, nam)
        return rslt
```

辞書とNVListの変換

OpenRTM-aistではOMGの標準仕様としては明確には決められていないけれど実はとても重要な情報をSDO(同じくOMGの標準)に倣ってNVList(NamedValueList)に保持しています。さらにvalueとして何

でも持てるようにcorbaのany型を使うという工夫をしています。このままだとpythonでは使いにくいので、ここではpythonの辞書型との変換･逆変換関数を定義して必要に応じて使うことにしています。

```
#
# data conversion
#
def nvlist2dict(nvlist) :
   rslt = {}
   for tmp in nvlist :
       rslt[tmp.name]=tmp.value.value()    # nv.value and any.value()
   return rslt
def dict2nvlist(dict) :
   rslt = []
   for tmp in dict.keys() :
       rslt.append(SDOPackage.NameValue(tmp, any.to_any(dict[tmp])))
   return rslt
```

Connectorクラスの定義

これは、RTC.ConnectorProfileに対応する重要なクラスです。ConnectorProfile自身は分散オブジェクトではありませんが、rtcのポートを接続するときに、各ポート間で ConnectorProfileをやりとりします。また接続時には、各ポートの実際に通信するオブジェクトの情報などが追記されていきます。これを利用して、Portオブジェクトを作るときにそのポートの情報を取得するためにそのポートだけを指定したコネクタを作って接続するということもやっています。実際の重要な情報はRTC.ConnectorProfileのpropertiesにNVListとして保持されます。コネクタが接続するのがサービスポートなのか入出力ポートなのかなどもpropertiesに記述されていて、RTC.ConnectorProfileの型では区別されません。 python上のConnectorクラスではpropertiesを辞書に変換し、また、ポートの種別ごとにサブクラスとすることで使い易くしています。サービス用と入出力用のコネクタの差はデフォールトのpropertiesの差になっています。

```
#
#  connector, port, inport, outport, service
#

class Connector :
   def __init__(self, plist, name = None, id="", prop_dict={}) :
      self.plist = plist
      self.port_reflist = [tmp.port_profile.port_ref for tmp in plist]
      if name :
          self.name = name
      else :
          self.name = string.join([tmp.name for tmp in plist],'_')
      self.prop_dict = prop_dict
      self.prop_nvlist = dict2nvlist(self.prop_dict)
      self.profile = RTC.ConnectorProfile(self.name, id, self.port_reflist, self.prop_
      self.nego_prop()
```

nego_propでは、デフォールトのpropertiesおよび接続するポートのproperties間で属性の調整をします。ここではポートのpropertiesレベルでの調整なので、実際の通信の型チェックなどは行いません。(実際の通信の型は一回は接続して見ないと分からない。)

```
    def nego_prop(self) :
        self.possible = True
        for kk in self.def_prop :
            if kk in self.prop_dict :
                if not self.prop_dict[kk] :
                    self.prop_dict[kk]=self.def_prop[kk]
            else :
                 self.prop_dict[kk]=self.def_prop[kk]
            for pp in self.plist :
                if not ((self.prop_dict[kk] in pp.prop[kk]) or
                                ('Any' in    pp.prop[kk])) :
                    self.prop_dict[kk] = ""
                    self.possible = False
        self.prop_nvlist = dict2nvlist(self.prop_dict)
        self.profile.properties = self.prop_nvlist
        return self.possible
```

connectでは、ポートリストの先頭のポートにconnectを発行します。残りのポートへは、そのポートがconnectを発行します。接続結果と最終的なポートプロファイルが戻ってきます。戻ってきたポートプロファイルを辞書形式に変換して保持しています。

```
    def connect(self) :
        ret, self.profile = self.port_reflist[0].connect(self.profile)
        self.prop_nvlist = self.profile.properties
        self.prop_dict = nvlist2dict(self.prop_nvlist)
        return ret
```

disconnectでは、connector_idを指定する必要があります。connector_idはconnectしたときに値が戻ってきています。

```
    def disconnect(self) :
        ret = self.port_reflist[0].disconnect(self.profile.connector_id)
        return ret
```

IOConnector, ServiceConnectorクラス定義

これらの差は、プロパティのみに表されています。

```
class IOConnector(Connector) :
    def __init__(self, plist, name = None, id="", prop_dict={}) :
        self.def_prop = {'dataport.dataflow_type':'Push',
                         'dataport.interface_type':'CORBA_Any',
                         'dataport.subscription_type':'Flush'}
        Connector.__init__(self, plist, name, id, prop_dict)

class ServiceConnector(Connector) :
```

```
    def __init__(self, plist, name = None, id="", prop_dict={}) :
        self.def_prop = {'port.port_type':'CorbaPort' }
        Connector.__init__(self, plist, name, id, prop_dict)
```

Portクラスの定義

ポートクラスもサービスポートと入出力ポートは形式上区別されません。PortProfie**の**properties**の内容で区別されます。とりあえずその**properties**を扱いやすいように辞書に変換します。**get_info**では、ポートに接続要求を出すことで、**PortProfile**の情報をもらっています。**get_info**の前には、各サブクラスでそのサブクラスに対応した**Connector**を生成し、**self.con**に割り当てておく必要があります。情報取得後、**disconnect**していますが、 その際に、**disconnect**の対象である接続がちゃんと記録されているかどうかをチェックしています。これにより接続が存在しないポートに**disconnect**を発行してしまい**RTC**を殺してしまうことを回避しています。接続情報は**get_connections()**を使うことで取得できます。**get_connections()**の戻り値は**ConnectorProfile**のリストになっています。**

```
class Port :
    def __init__(self, profile,nv_dict=None) :
        self.name=profile.name
        self.port_profile = profile
        if not nv_dict :
            nv_dict = nvlist2dict(profile.properties)
        self.prop = nv_dict
        self.con = None           # this must be set in each subclasses
    def get_info(self) :
        self.con.connect()
        tmp1 = self.get_connections()
        tmp2 = [pp.connector_id for pp in tmp1]
        if self.con.profile.connector_id in tmp2 :
            self.con.disconnect()
    def get_connections(self) :
        return self.port_profile.port_ref.get_connector_profiles()
```

RtcServiceクラスの定義とCorbaServer、CorbaClient

1つのサービスポートは、複数のserver**と**client**で構成されます。ここでは**Corba**で実装されたものを扱うことにします。 server**のオブジェクトリファレンスは、サービスポートを**connect**を要求したときに**ConnectorProfile**の**properties**に入って戻ってきています。**

NVListにおける名前(ref_key)は、port.*クラス名(type_name).サービス名(instance_name)*になっています。オブジェクトリファレンスをserverのクラスにnarrowするためには、 serverクラスのスタブをimportする必要があります。 pythonの場合、事前にimportしておけば、自動的にnarrowされるようです。

```
class CorbaServer :
    def __init__(self, profile, port) :
        self.profile = profile
        self.port = port
        self.name = profile.instance_name
```

```
        self.type = profile.type_name
        self.ref = None
        ref_key = 'port.' + self.type + '.' + self.name
        self.ref=self.port.con.prop_dict[ref_key]

    def narrow_ref(self, gls) :
        if self.type.find('::') == -1 :
            self.narrow_sym = eval('_GlobalIDL.' + self.type, gls)
        else :
            self.narrow_sym = eval(self.type.replace('::','.'), gls)
        self.ref = self.ref._narrow(self.narrow_sym)
```

clientを使うためには、スケルトンをimportし実装を作成しなくていけません。ここでは、とりあえず扱わないことにします。

```
class CorbaClient :
    def __init__(self, profile) :
        self.profile = profile
        self.name = profile.instance_name
        self.type = profile.type_name
#
# to connect to an outside corba client,
# we need an implementation of the corresponding corba server.
# but ....
#
```

RtcServiceクラスはPortのサブクラスです。 get_infoでは、ポートに接続要求を出してPortProfileに情報をもらっています。それをもとにserver(provided)、client(required)の情報を整理しています。 get_infoをやる前には、self.conに自分自身だけに対するServiceConnectorを割り当てておく必要があります。 サービスポートのサーバを使うには、

env.rtc_handles[*"rtc_name"*].services[*"port_name"*].provided[*"server_name"*].ref.*operation(...)*

などとすることになります。もちろん事前に、

aho = *env*.rtc_handles[*"rtc_name"*].services[*"port_name"*].provided[*"server_name"*].ref

などとしておけば、

aho.*operation(...)*

というように簡単に使えます。

```
class RtcService(Port) :
    def __init__(self, profile,nv_dict=None) :
        Port.__init__(self, profile, nv_dict)
        self.con = ServiceConnector([self])
        self.get_info()
```

```
        self.provided={}
        self.required={}
        tmp = self.port_profile.interfaces
        for itf in tmp :
            if itf.polarity == RTC.PROVIDED :
                self.provided[itf.instance_name] = CorbaServer(itf,self)
            elif itf.polarity == RTC.REQUIRED :
                self.required[itf.instance_name] = CorbaClient(itf)
```

RtcInportのクラス定義

RtcServiceと同様にget_infoをやる前には、self.conに自分自身だけに対するIOConnectorを割り当てておく必要があります。

```
class RtcInport(Port) :
   def __init__(self, profile, nv_dict=None) :
       Port.__init__(self, profile, nv_dict)
       self.con = IOConnector([self])
       self.get_info()
       self.ref = self.con.prop_dict['dataport.corba_any.inport_ref']
       self.data_class = eval('RTC.' + self.prop['dataport.data_type'])
       self.data_tc = eval('RTC._tc_' + self.prop['dataport.data_type'])
   def write(self,data) :
       self.ref.put(CORBA.Any(self.data_tc,
                              self.data_class(RTC.Time(0,0),data)))
```

OutPortのクラス定義

get_infoをやる前には、self.conに自分自身だけに対するIOConnectorを割り当てておく必要があるのは他のPortと同じです。少し複雑になっているのは、'dataport.corba_any.outport_ref'の値を返さないJAVA版のバグ回避のためです。

```
class RtcOutport(Port) :
   def __init__(self, profile,nv_dict=None) :
       Port.__init__(self, profile, nv_dict)
       self.con = IOConnector([self])
       self.get_info()
       if 'dataport.corba_any.outport_ref' in self.con.prop_dict :
          self.ref = self.con.prop_dict['dataport.corba_any.outport_ref']
       else :
          self.ref=None
       self.data_class = eval('RTC.' + self.prop['dataport.data_type'])
       self.data_tc = eval('RTC._tc_' + self.prop['dataport.data_type'])
   def read(self) :
       if self.ref :
          return self.ref.get().value()
```

```
        else :
           print "not supported"
           return None
```

RtcHandleのクラス定義

__init__

とりあえず名前、環境、RTCのオブジェクトリファレンスを保持して retreieve_infoへ

```
#
# RtcHandle
#
class RtcHandle :
 def __init__(self, name, env, ref=None) :
   self.name = name
   self.env = env
   if ref :
       self.rtc_ref = ref
   else :
       self.rtc_ref = env.naming.resolve(name)._narrow(RTC.RTObject)
   self.configuration_ref = None
   self.retrieve_info()
```

retrieve_info

まずはconfiguration serviceのオブジェクトレファレンスを取得して、configuration setのリストを辞書形式でself.conf_set保持します。各configuration setのデータは、NVlist形式なので、辞書形式に変換したものを別途self.conf_set_datに保持します。configuration setのパラメタの値を変更するときは、まず辞書の方を 変更してconfiguration setに反映させます。

```
 def retrieve_info(self) :
   self.conf_ref = self.rtc_ref.get_configuration()
   self.conf_set={}
   self.conf_set_data={}
   self.port_refs = []
   self.execution_contexts =[]
   if self.rtc_ref :
      self.conf_ref = self.rtc_ref.get_configuration()
      conf_set = self.conf_ref.get_configuration_sets()
      for cc in conf_set :
          self.conf_set[cc.id]=cc
          self.conf_set_data[cc.id]=nvlist2dict(cc.configuration_data)
```

ポートは、get_component_profileにも情報があるはずなのですが、今は情報が完全ではありません。get_portsでポートリストを取得して、ポート自身からport profileを取得します。ポートの種別は、port profileのpropertiesを見なくては分かりません。

```
        self.profile = self.rtc_ref.get_component_profile()
        self.prop = nvlist2dict(self.profile.properties)
        self.execution_contexts = self.rtc_ref.get_contexts()
        self.port_refs = self.rtc_ref.get_ports()
                 # this includes inports, outports and service ports
    self.ports = {}
    self.services = {}
    self.inports = {}
    self.outports = {}
    for pp in self.port_refs :
        tmp = pp.get_port_profile()
        tmp_prop = nvlist2dict(tmp.properties)
#       self.ports[tmp.name]=Port(tmp, tmp_prop)
        if tmp_prop['port.port_type']=='DataInPort' :
            self.inports[tmp.name]=RtcInport(tmp,tmp_prop)
#             self.inports[tmp.name]=Port(tmp, tmp_prop)
        elif tmp_prop['port.port_type']=='DataOutPort' :
            self.outports[tmp.name]=RtcOutport(tmp, tmp_prop)
#             self.outports[tmp.name]=Port(tmp, tmp_prop)
        elif tmp_prop['port.port_type']=='CorbaPort' :
            self.services[tmp.name]=RtcService(tmp, tmp_prop)
#             self.services[tmp.name]=Port(tmp, tmp_prop)
```

configurationのパラメタの変更には、configuration set**名とparameter名を指定します。** value**は文字列表現であることに注意してください。** interface**としては、他にもいろいろな設定方法がありますが、分かり易さの点で これ1つにしています。下請けはでは、**configuration set**ごと入れかえる** operation**を呼び出しています。パラメタ変更を、相手の**RTC**の内部に反映させるには、**activate_configuration_set**を呼び出す必要があります。** configuration set**の削除、追加などの** operation**については、**OpenRTM-aist**のIDLリファレンスの**SDOPackage.idl**を見て下さい。そこで記述された**operation**は、**conf_ref.*operation()*として呼び出すことができます。ただし現状の**OpenRTM-aist-0.4.2**では全部が正しく実装されているとは限りません。**

set_conf, set_conf_activate

```
  def set_conf(self,conf_set_name,param_name,value) :
      conf_set=self.conf_set[conf_set_name]
      conf_set_data=self.conf_set_data[conf_set_name]
      conf_set_data[param_name]=value
      conf_set.configuration_data=dict2nvlist(conf_set_data)
      self.conf_ref.set_configuration_set_values(conf_set_name,conf_set)
  def set_conf_activate(self,conf_set_name,param_name,value) :
      self.set_conf(conf_set_name,param_name,value)
      self.conf_ref.activate_configuration_set(conf_set_name)
```

activate, deactivate

```
  def activate(self):
      self.execution_contexts[0].activate_component(self.rtc_ref)
  def deactivate(self):
      self.execution_contexts[0].deactivate_component(self.rtc_ref)
```

Pipeクラス

Pipeクラスは作りかけです。

python環境を一つのRTCとして実現し、そのポートを作成して他のRTCにアクセスするメカニズムを作ることを考えています。

```
#
# pipe
#  a pipe is an port (interface & implementation)
#  whhich corresponds to an outside port interface.
#  you can subscribe and communicate to the outside port with the pipe.
#  you need an rtc implementation to use pipes.
#
class Pipe :
   pass

class PipeOut(Pipe):
   def __init__(self, name, data_buf, size=8) :
       self.name =  name
       self.data =  data_buf
       self.OpenRTM.InPort(name,data_buf,OpenRTM.RingBuffer(size))
```

各クラスの構造

こういうものを表示する良いツールがあるといいのだけど。doxygenでは、こういう図にはならないのです。本来は、UMLのクラス図でしょうか。

RtmEnv

```
env -+- orb
     +- namespace[ns_name]
```

NameSpace

```
n_space -+- name
         +- orb
         +- naming
         +- b_len
```

```
        +- rtc_handles[h_name]
        +- obj_list[o_name]
        |
        +- get_objct_by_name(name, cl=obj_class)
        +- list_obj()
        +- list_obj1(...)
        +- proc_bd(...)
```

RtcHandle

```
handle -+- name
        +- env                # RtmEnv
        +- rtc_ref            # env.naming.resolve(name)._narrow(RTC.RTObject)
        +- conf_ref           # configuration serviceのオブジェクトリファレンス
        +- conf_set           # configuration set の辞書
        +- conf_set_data      # 上記のconfiguration_data (NVList) の辞書表現
        +- port_refs[idx]     # self.rtc_ref.get_ports()
        +- execution_contexts[idx] # self.rtc_ref.get_contexts()
        +- profile            # self.rtc_ref.get_component_profile()
        +- port[p_name]       # 未使用。過去の残骸。以下３つと冗長。
        +- services[s_name]   # サービスポート(RtcService)
        +- inports[i_name]    # 入力ポート(RtcInport)
        +- outports[oname]    # 出力ポート(RtcOutport)
        |
        +- retrieve_info()    # 各Profileの取得と各ポートの生成
        +- set_conf(conf_set_name, param_name, value)
        +- set_conf_activate(conf_set_name, param_name, value)
        +- activate()
        +- deactivate()
```

参考：Windows XPでの使用例

私はwindows版はあまりつかわないので取り合えず試しただけです。

Python command line 版

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> RtmToolsDir = "c:¥users¥openrtm¥python"
>>> sys.path
['', 'C:¥¥WINDOWS¥¥system32¥¥python25.zip', 'C:¥¥Python25¥¥DLLs', 'C:¥¥Python25¥
¥lib', 'C:¥¥Python25¥¥lib¥¥plat-win', 'C:¥¥Python25¥¥lib¥¥lib-tk', 'C:¥¥Python25
', 'C:¥¥Python25¥¥lib¥¥site-packages', 'C:¥¥Python25¥¥lib¥¥site-packages¥¥OpenRT
```

```
M', 'C:¥¥Python25¥¥lib¥¥site-packages¥¥OpenRTM¥¥RTM_IDL']
>>> save_path=sys.path[:]
>>> sys.path.append(RtmToolsDir)
>>> from rtc_handle import *
>>> sys.path=save_path
>>> RtmEnv
<class rtc_handle.RtmEnv at 0x01225960>
>>> env=RtmEnv(sys.argv, ["192.168.136.129:9876"])
>>> env.name_space["192.168.136.129:9876"].list_obj()
objcet move0.rtc was listed.
handle for move0.rtc was created.
objcet EmptyRtc0_py.rtc was listed.
handle for EmptyRtc0_py.rtc was created.
objcet SampleRevRtc0_py.rtc was listed.
handle for SampleRevRtc0_py.rtc was created.
objcet SampleRtc0_py.rtc was listed.
handle for SampleRtc0_py.rtc was created.
objcet tr_jacob0.rtc was listed.
handle for tr_jacob0.rtc was created.
objcet coord_trans0.rtc was listed.
handle for coord_trans0.rtc was created.
objcet jinv0.rtc was listed.
handle for jinv0.rtc was created.
objcet vel_7dof0.rtc was listed.
handle for vel_7dof0.rtc was created.
objcet j_jacob0.rtc was listed.
handle for j_jacob0.rtc was created.
objcet frm_ctrl0.rtc was listed.
handle for frm_ctrl0.rtc was created.
objcet pa10fk0.rtc was listed.
handle for pa10fk0.rtc was created.
objcet mixer0_py.rtc was listed.
handle for mixer0_py.rtc was created.
objcet sliderComp0.rtc was listed.
handle for sliderComp0.rtc was created.
objcet pa10disp0.rtc was listed.
handle for pa10disp0.rtc was created.
[['move0.rtc', <RTC._objref_DataFlowComponent instance at 0x0122E288>], ['EmptyR
tc0_py.rtc', <RTC._objref_DataFlowComponent instance at 0x01243DC8>], ['SampleRe
vRtc0_py.rtc', <RTC._objref_DataFlowComponent instance at 0x012560D0>], ['Sample
Rtc0_py.rtc', <RTC._objref_DataFlowComponent instance at 0x0125D0F8>], ['tr_jaco
b0.rtc', <RTC._objref_DataFlowComponent instance at 0x01262B20>], ['coord_trans0
.rtc', <RTC._objref_DataFlowComponent instance at 0x0126E288>], ['jinv0.rtc', <R
TC._objref_DataFlowComponent instance at 0x01271D28>], ['vel_7dof0.rtc', <RTC._o
bjref_DataFlowComponent instance at 0x012A81C0>], ['j_jacob0.rtc', <RTC._objref_
DataFlowComponent instance at 0x012B00A8>], ['frm_ctrl0.rtc', <RTC._objref_DataF
```

```
lowComponent instance at 0x012B45D0>], ['pa10fk0.rtc', <RTC._objref_DataFlowComp
onent instance at 0x012C19B8>], ['mixer0_py.rtc', <RTC._objref_DataFlowComponent
 instance at 0x012C8F80>], ['sliderComp0.rtc', <RTC._objref_DataFlowComponent in
stance at 0x012D0D00>], ['pa10disp0.rtc', <RTC._objref_DataFlowComponent instanc
e at 0x012D4490>]]
>>>
```
