ＲＴＭによるコンポーネント化作業
「学習・推論コンポーネント」
開発手順書

平成 19 年 05 月 30 日

株式会社アドイン研究所

AdIn Research, Inc.

【目次】

１．はじめに

RT ミドルウェアは、RT コンポーネントを開発するためのフレームワークを提供している。このフレームワークを利用すると、既存のソフトウェア資産を、容易に RT コンポーネント化することができる。

本書は、株式会社アドイン研究所のファジィニューロ学習・推論エンジンであるβ-RNA を、RT ミドルウェア OpenRTM-aist-0.4.0 が提供する、RT コンポーネント化のための開発フレームワークを利用して、RT コンポーネント化したときの開発手順を示したものである。

２．開発手順

β-RNA を、RT ミドルウェア OpenRTM-aist-0.4.0 の RT コンポーネント化のための開発フレームワークを用いて、RT コンポーネント化したときの作業手順は、以下の通りである。

・検討作業
学習・推論コンポーネントの機能や使い方を検討、決定する。

・設計作業
学習・推論コンポーネントの詳細（外部仕様）を検討、決定する。

・実装作業
学習・推論コンポーネントを実装する。

・試験作業
学習・推論コンポーネントを試験する。

３．検討作業

β-RNA は、株式会社アドイン研究所のファジィニューロ学習・推論エンジンである。β-RNA は、C 言語で実装されており、ライブラリの形態で提供されている。β-RNA を、RT コンポーネント化して、β-RNA コンポーネントを開発するにあたり、まず、以下の項目を検討した。

検討事項１：　β-RNA コンポーネントで、何を実現すればよいか？
検討事項２：　β-RNA コンポーネントは、どのような使い方をされるのか？

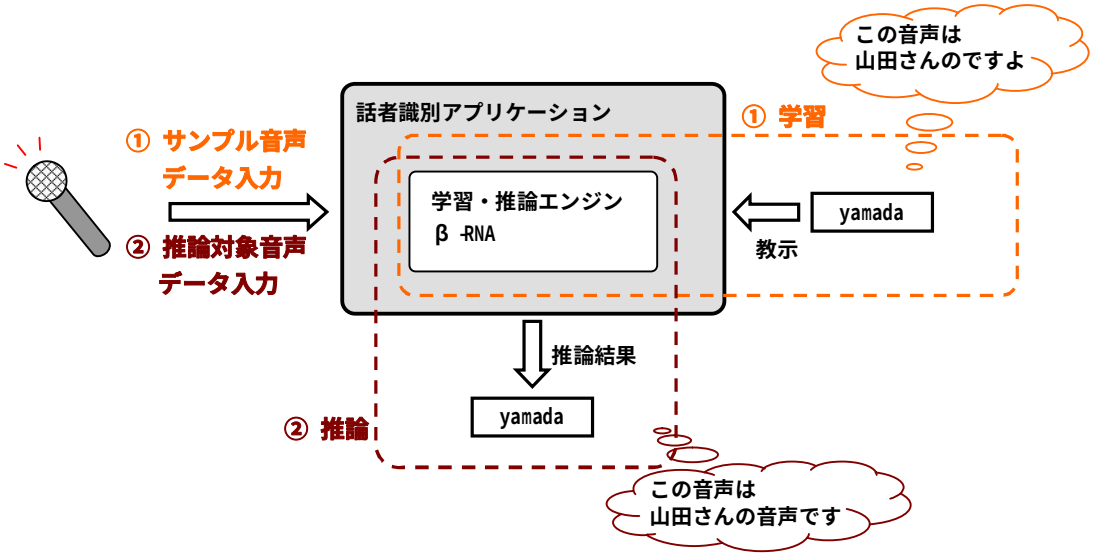
３.１．β-RNA で何ができるか？

検討事項１「β-RNA コンポーネントで、何を実現すればよいか？」を検討する前に、まず、「β-RNA は、何ができるのか？」を明確にする。まず、β-RNA がどのようなものであるかを示すため、β-RNA を用いたアプリケーションの例を紹介する。

下図は、β-RNA を用いた話者識別アプリケーションの概略である。このアプリケーションは、２つの機能を持っている。

機能１：アプリケーションに音声データと、その音声データの発話者の名称を入力する。この操作を繰り返すことで、アプリケーションは、発話者の音声の特徴を抽出して、学習する。（＝**学習機能**）

機能２：アプリケーションに、新しい音声データを入力すると、アプリケーションは学習機能で蓄積した学習データを元に、「その音声データの発話者は誰か？」を推論（音声データの特徴が、誰の音声の特徴に最も近いかを計算する）して、その結果を出力する。（＝**推論機能**）



この、話者識別アプリケーションの学習機能、推論機能は、β-RNA により実現される（β-RNA が提供する）機能である。つまり、β-RNA を用いることで、学習機能、推論機能を必要とするアプリケーションが、容易に開発できるようになる。

３.２. 検討作業

３.２.１. 検討項目１

検討項目１「β-RNA コンポーネントで、何を実現すればよいか？」を検討する。既存のソフトウェアライブラリであるβ-RNA の機能は、学習機能と、推論機能ということであった。従って、検討項目１の検討結果は、学習機能と、推論機能とを実現できればよいと考えられる。

> 「β-RNA コンポーネントで、何を実現すればよいか？」
> ⇒ 学習機能と、推論機能とを実現すればよい。

３.２.２. 検討事項２

次に、検討事項２「β-RNA コンポーネントは、どのような使い方をされるのか？」について検討する。これは、β-RNA の学習機能がどのような場合に、利点となるのかを考えることが（つまり、既存のソフトウェアがどのような利点を持っているのかを考えることが）、役に立った。

β-RNA の学習機能は、「推論のためのルールを、人が容易に記述することができない」ような場合に、利点となる。例えば、簡単な線形関数を使って、推論するためのルールを記述できる場合、あるいは、単純な IF～ELSE～THEN を使って、推論するためのルールを記述できる場合は、β-RNA の学習機能は、利点になるとはいえない（β-RNA の学習機能を利用するよりも、人が、ルールを記述したほうが早い）。これに対して、話者識別、画像認識等を行う場合、推論（識別、認識）のためのルールを、人が記述することは非常に難しい。このような場合、β-RNA の学習機能を使用することは、大きな利点となる。従って、検討事項２の検討結果は、話者識別、画像認識等、「推論（識別、認識）のためのルールを、人が記述することが難しい」ような場合の推論を行うためのエンジンとして、使われるということになる。

> 「β-RNA コンポーネントは、どのような使い方をされるのか？」
> ⇒ 話者識別、画像認識等、「推論（識別、認識）のためのルールを、人が記述することが難しい」ような場合の推論を行うためのエンジンとして使われる。

以上の検討で、β-RNA コンポーネントの機能、役割が明確になった。従って、以後、β-RNA を、RT コンポーネント化したものを、「**学習・推論コンポーネント**」と呼ぶことにする。

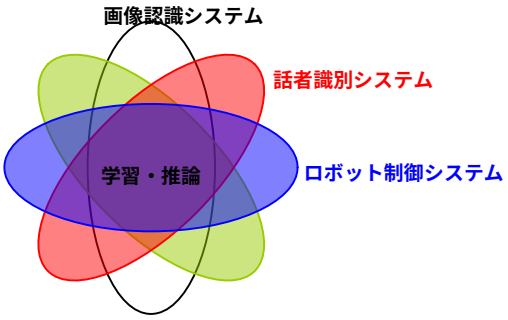
４．設計作業

本章では、学習・推論コンポーネントの外部仕様（インターフェース）を検討する。

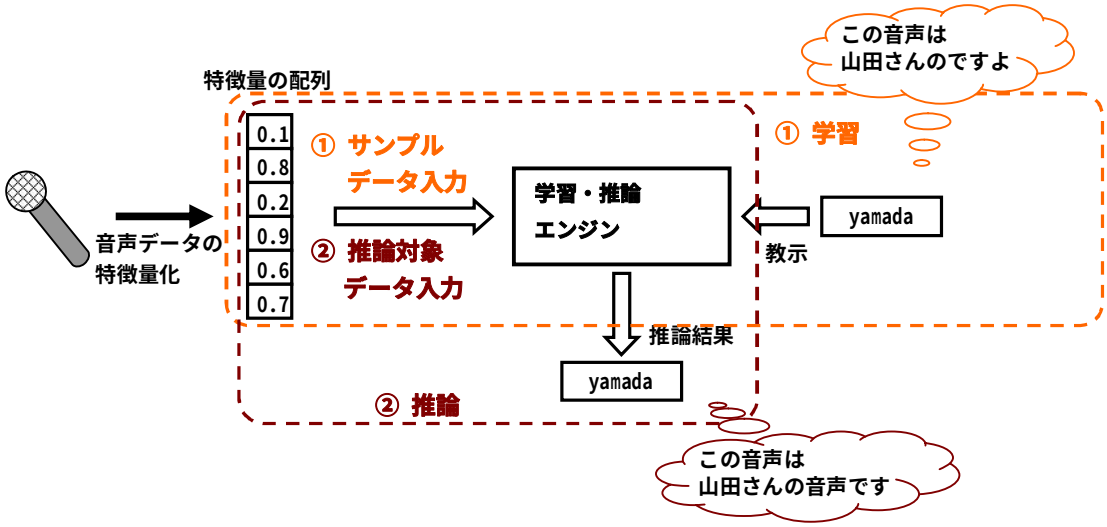
４.１.１．コンポーネントのインターフェースの決定

オブジェクト指向プログラミングにおいて、オブジェクトのインターフェースを検討する際は、「オブジェクトの粒度をどうするか？」ということを念頭に置くと、見通しが良くなる。これに倣い、「コンポーネントの粒度をどうするか？」ということを念頭に置いて、学習・推論コンポーネントのインターフェースを検討した。

学習・推論コンポーネントは、「推論のためのルールを、人が記述することが難しい」ようなシステム、つまり、画像認識システム、話者識別システム、ロボット制御システム等の、学習・推論を担うコンポーネントとして利用されると思われる。従って、学習・推論コンポーネントは、これらのシステムから、汎用的に利用できる粒度でなければならない。

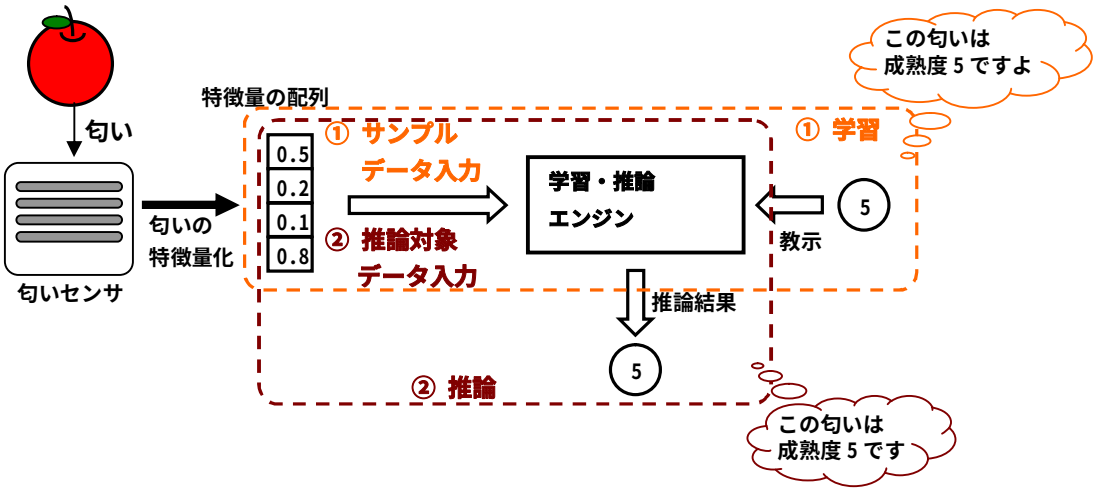


ここで、学習・推論エンジンを利用したシステムの仕組みを考える。まず、学習・推論エンジンを用いた代表的なシステムである、話者識別システムの仕組みを下図に示す。



この場合、学習・推論エンジンへの入力は、音声の特徴量の集合（＝数値配列）、出力は、名称（＝文字列）となる。

次に、学習・推論エンジンを利用した、変わった例である、匂いセンサによる果実の成熟度評価システムの仕組みを、下図に示す。



この場合、学習・推論エンジンへの入力は、匂いの特徴量の集合（＝数値配列）、出力は、成熟度（＝整数）となる。

両システムの、学習・推論エンジンへの入出力データを、比較してみると、下表のようになる。

システム	学習・推論エンジンへの入力データ	学習・推論エンジンからの出力データ
話者識別システム	特徴量の集合（数値配列）　(*)	氏名（文字列）
匂いセンサによる果実の成熟度評価システム	特徴量の集合（数値配列）　(*)	成熟度（整数）

(*)学習・推論エンジンには、教示のための入力データも存在するが、これは、出力データと同じ形式のデータとなるので、割愛する。)

この比較より、以下の結論が得られる。

1. システムへの入力データ（生データ）を特徴量化する処理は、システムにより異なる。従って、学習・推論コンポーネントの内部に、特徴量化する処理は組み込まない（特徴量化を担当するコンポーネントを、別で用意すればよい）。
2. 学習・推論エンジンには、共通して、特徴量の集合が入力される。つまり、学習・推論コンポーネントの入力インターフェースは、数値配列とすれば、汎用性が高い。
3. 学習・推論エンジンの出力データは、識別を目的とするデータ（＝氏名）と、比較を目的とするデータ（＝成熟度）とが考えら得れる。ここで、識別は、氏名でなくても、ID（整数）を用いて行うことができる（外部に、ID 対氏名のデータベースを用意すれば、システムの出力値を、氏名とすることができる）。従って、学習・推論コンポーネントの出力データは、整数とすれば、汎用性が高い。

以上により、学習・推論コンポーネントのインターフェースは、以下の通りとした。

学習・推論コンポーネントへの入力データ	**数値配列** **整数（＝教示データ。出力データと同一形式）**
学習・推論コンポーネントへの出力データ	**整数**

５．実装作業

前章の検討の結果を踏まえて、学習・推論コンポーネントのインターフェースを、以下のように決定した。

入力データポート（InPort）仕様

ポート名称	データ型	説明
ID	TimedShort	本ポートで、動作モード（学習モード/推論モード）の切り替えを行う。学習モード時は、教示のために、サンプルデータのグループを指定する役割も併せ持つ。 (学習モード) サンプルデータのグループ ID (推論モード) -1
ParamDir	TimedString	データディレクトリ（学習パラメタファイル、学習結果ファイルを格納するディレクトリ）を指定する。なお、データディレクトリは、コンポーネントサーバに存在する必要がある。
DataCount	TimedShort	学習モード時はサンプルデータの、推論モード時は、判別対象データの、**特徴量の項目数**を指定する。
DataAry	TimedDoubleSeq	学習モード時はサンプルデータの、推論モード時は、判別対象データの、**特徴量の配列**を指定する。各特徴量は、**0 以上 1 以下**でなければならない。

出力データポート（OutPort）仕様

ポート名称	データ型	説明
Result	TimedShort	推論モード時は、推論結果のグループ ID が出力される。 （学習モード時は、入力ポート ID で指定した、サンプルデータのグループ ID が、そのまま出力される。）
Trusty	TimedDouble	推論モード時のみ合致度（推論結果が信頼できる度合い、**0 以上 1 以下**）を出力する。（学習モード時は、 -1 が出力される。）

５.１. コンポーネント雛形の作成

RT ミドルウェア OpenRTM-aist には、RT コンポーネントの雛形を、コードジェネレータ rtc-template を使って、自動的に生成する仕組みが存在する。この仕組みを使って、学習・推論コンポーネントの雛形を生成した。

下に、学習・推論コンポーネントの雛形を生成するために用いた、rtc-template 起動シェルスクリプトを示す。

```
#!/bin/bash
rtc-template -bcxx \
  --module-name=BetaRnaPort --module-type='BetaRnaPort' \
  --module-desc='BetaRna in/outport component' \
  --module-version=0.1 --module-category=Generic \
  --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
  --module-max-inst=10 \
  --inport=DataCount:TimedShort --inport=DataAry:TimedDoubleSeq \
  --inport=ID:TimedShort --inport=ParamDir:TimedString \
  --outport=ResultID:TimedShort --outport=Trusty:TimedDouble
```

[rtc-template のオプションについて]

一般的なオプション

-bcxx は C++で RT コンポーネント開発を行う場合は、指定で必須である。

--module-name、--module-type は RT コンポーネント名称を設定する（必須)。

--module-category、--module-comp-type、--module-act-type は一般的な設定値とした。

rtc-template の詳細については、下の URL を参照すること。

http://www.is.aist.go.jp/rt/OpenRTM-aist-Tutorial/programming2.html

データポートの指定方法

コードジェネレータ rtc-template のオプションで、データポート（Inport/Outport）の定義を指定できる（上の rtc-template 起動シェルスクリプトの太字部)。このオプションの書式を、以下に示す。

```
入力データポートの指定　：　--inport=変数名:型
出力データポートの指定　：　--outport=変数名:型
```

rtc-template 起動シェルスクリプトを実行すると、「m_"指定された変数名"」という名称の、入力データポートへの入力値 / 出力データポートへの出力値を扱うための変数が、RT コンポーネントクラスのメンバ変数として、自動的に定義される。下に、rtc-template により生成された、学習・推論コンポーネントの雛形を示す。赤い文字列が、自動的に定義された、データポートの入出力値を扱うための変数である。

```
class BetaRnaPort
  : public RTC::DataFlowComponentBase
{
public:
  BetaRnaPort(RTC::Manager* manager);
  ~BetaRnaPort();

  ...(略)...

protected:
  // DataInPort declaration
  // <rtc-template block="inport_declare">
  TimedShort m_DataCount;
  InPort<TimedShort> m_DataCountIn;
  TimedDoubleSeq m_DataAry;
  InPort<TimedDoubleSeq> m_DataAryIn;
  TimedShort m_ID;
  InPort<TimedShort> m_IDIn;
  TimedString m_ParamDir;
  InPort<TimedString> m_ParamDirIn;
  // </rtc-template>

  // DataOutPort declaration
  // <rtc-template block="outport_declare">
  TimedShort m_ResultID;
  OutPort<TimedShort> m_ResultIDOut;
  TimedDouble m_Trusty;
  OutPort<TimedDouble> m_TrustyOut;
  // </rtc-template>

  ...(略)...
};
```

なお、「m_XXXXIn」、「m_XXXXOut」という変数も、自動的に定義される。これは、データポートを操作するための変数であり、以下の役割を持つ。

変数名称	説明	使用例
m_XXXXIn	入力データポートの操作を行う。 ＝入力データポートに入力されたデータを取得する。	m_XXXXIn.Read() => m_XXXX に、入力データポートに入力されたデータが入る。
m_XXXXOut	出力データポートの操作を行う。 ＝出力データポートを通して、他のコンポーネントに、データを出力する。	m_XXXXIn.Write(); => m_XXXX に設定されている値が、出力データポートから出力される。

５.２. 処理の作成

５.２.１. 実装関数の決定

rtc-template 起動シェルスクリプトを実行により生成された雛形には、RT コンポーネントの状態に応じて呼び出される関数が、既に用意されている。これらの関数は、デフォルトでは、コメントアウトされているので、必要に応じて、関数をコメントインして、各コンポーネントに固有の処理を記述することになる。

通常は、以下の３つをコメントインして、各コンポーネントに固有の処理を記述すればよい。

```
// コンポーネントがアクティブになったときに呼ばれる。
virtual RTC::ReturnCode_t onActivated(RTC::UniqueId ec_id);

// コンポーネントが非アクティブになったときに呼ばれる。
virtual RTC::ReturnCode_t onDeactivated(RTC::UniqueId ec_id);

// コンポーネントが実行されたときに呼ばれる。以後、定期的（非常に短周期）で呼ばれる。
virtual RTC::ReturnCode_t onExecute(RTC::UniqueId ec_id);
```

学習・推論コンポーネントでは、コンポーネントが非アクティブになったときに、特別な処理を行う必要が無かったため、onActivated 関数と、onExecute 関数のみ、コメントインして、処理を記述した。

まず、ヘッダファイルで、onActivated 関数と、onExecute 関数をコメントインした（下の太線部）。

```
class BetaRnaPort
  : public RTC::DataFlowComponentBase
{
  ...(略)...

  // The initialize action (on CREATED ->ALIVE transition)
  // formaer rtc_init_entry()
  // virtual RTC::ReturnCode_t onInitialize();

  ...(略)...
  // The activated action (Active state entry action)
  // former rtc_active_entry()
  virtual RTC::ReturnCode_t onActivated(RTC::UniqueId ec_id);

  // The deactivated action (Active state exit action)
  // former rtc_active_exit()
  // virtual RTC::ReturnCode_t onDeactivated(RTC::UniqueId ec_id);

  // The execution action that is invoked periodically
  // former rtc_active_do()
  virtual RTC::ReturnCode_t onExecute(RTC::UniqueId ec_id);

  // The aborting action when main logic error occurred.
  // former rtc_aborting_entry()
  // virtual RTC::ReturnCode_t onAborting(RTC::UniqueId ec_id);

  ...(略)...
};
```

次に、実装ファイル（cpp ファイル）で、onActivated 関数と、onExecute 関数をコメントインした（下の太線部）。

```
/*
RTC::ReturnCode_t BetaRnaPort::onShutdown(RTC::UniqueId ec_id)
{
  return RTC::RTC_OK;
}
*/

RTC::ReturnCode_t BetaRnaPort::onActivated(RTC::UniqueId ec_id)
{
  return RTC::RTC_OK;
}

/*
RTC::ReturnCode_t BetaRnaPort::onDeactivated(RTC::UniqueId ec_id)
{
  return RTC::RTC_OK;
}
*/

RTC::ReturnCode_t BetaRnaPort::onExecute(RTC::UniqueId ec_id)
{
  return RTC::RTC_OK;
}
```

５.２.２. 入力データポート操作変数を、RTC::RingBuffer の派生クラスに変更

入力データポート操作変数(m_XXXXIn)を、RTC::RingBuffer の派生クラスに変更すると、入力ポートの更新を調べることが可能となる。

```
// ポートに更新があったかどうかを調べる関数
  bool RTC::RingBuffer::isNew()
```

入力データポート操作変数(m_XXXXIn)を、RTC::RingBuffer の派生クラスに変更するには、入力ポート操作変数の定義部に、以下の変更（下の赤文字）を加えればよい。

```
// 変更前（m_XXXXIn は、RTC::RingBuffer の派生クラスではない）
InPort<TimedLong> m_XXXXIn;

// 変更後（m_XXXXIn は、RTC::RingBuffer の派生クラスである）
InPort<TimedLong,RTC::RingBuffer>  m_XXXXIn;
```

学習・推論コンポーネントでは、４つの入力データポート操作変数を、RTC::RingBuffer の派生クラスに変更した（下の赤文字）。

```
class BetaRnaPort
  : public RTC::DataFlowComponentBase
{
  ...(略)...

protected:
  // DataInPort declaration
  // <rtc-template block="inport_declare">
  TimedShort m_DataCount;
  InPort<TimedShort,RTC::RingBuffer> m_DataCountIn;
  TimedDoubleSeq m_DataAry;
  InPort<TimedDoubleSeq,RTC::RingBuffer> m_DataAryIn;
  TimedShort m_ID;
  InPort<TimedShort,RTC::RingBuffer> m_IDIn;
  TimedString m_ParamDir;
  InPort<TimedString,RTC::RingBuffer> m_ParamDirIn;
  // </rtc-template>

  ...(略)...
};
```

５.２.３. 処理の記述

onActivated 関数、onDeactivated 関数、onExecute 関数の実装を行う。これらの関数は、一般的に、以下の通りに、処理を記述すればよい。

```
RTC::ReturnCode_t MyComponent::onActivated(RTC::UniqueId ec_id)
{
  m_XXXXIn.read();
  // RT コンポーネントに組み込むライブラリの、初期化処理等を記述
  return RTC::OK;
}

RTC::ReturnCode_t MyComponent::onDeactivated(RTC::UniqueId ec_id)
{
  // RT コンポーネントに組み込むライブラリの、終了処理等を記述
  return RTC::OK;
}

RTC::ReturnCode_t MyComponent::onExecute(RTC::UniqueId ec_id)
{
                    RTC::Buffer 派生クラスの変数とする必要がある
  if( m_XXXXIn.isNew() ) { // 入力データポートが更新された場合
    m_XXXXIn.read();       // 入力データポートの値を、m_XXXX 変数に読み込む

    // RT コンポーネントに組み込むライブラリを用いて、処理を記述する

    m_XXXXOut.write();     // m_XXXX 変数値を、出力データポートに書き出す
  }
  return RTC::OK;
}
```

なお、RTC::RingBuffer の派生クラスとしている入力データポート操作変数(m_XXXXIn)については、onActivated 関数内で、一度、read 関数を実行することを薦める。これは、起動直後に、入力データポートの入力値を扱う変数 m_XXXX に、ゴミが入っている（＝初期化されていない状態である）場合があるための処置である。

学習・推論コンポーネントにおいては、onActivated 関数、onExecute 関数に、以下の通りの実装を行った。なお、m_cBD は、β-RNA ライブラリの、単純なラッパークラスである。

```
RTC::ReturnCode_t BetaRnaPort::onActivated(RTC::UniqueId ec_id)
{
  m_DataCountIn.read(); // 入力ポートのごみを除く
  m_DataAryIn.read();   // 入力ポートのごみを除く
  m_IDIn.read();        // 入力ポートのごみを除く
  m_ParamDirIn.read();  // 入力ポートのごみを除く
  return RTC::OK;
}
```

```
RTC::ReturnCode_t BetaRnaPort::onExecute(RTC::UniqueId ec_id)
{
  static bool bTalker = false;
  static bool bParam = false;
  static int  iUserID =  -1;
  static int  iCount =  -1;
  int         iRet;
  double      dTrust;

  // ---------------------------------------------------------------
  // 話者 ID データ入力ポートに、新しいデータが入力された場合
  // ---------------------------------------------------------------
  if( m_IDIn.isNew() ) {
    // 話者 ID データ入力ポートから、データを読み込む
    m_IDIn.read();

    // 話者 ID データが、一度以上入力されたことを示す
    bTalker = true;

    // 話者 ID を保持
    iUserID = m_ID.data;

    // 動作モードを表示する
    if( iUserID < 0 ) {
      // 話者 ID が負数の場合は、推論モードで動作
      fprintf( stdout, "Set mode to detect.\n" );
    }
    else {
      // 話者 ID が負数の場合は、学習モードで動作
      fprintf( stdout, "Set mode to learn, output = %d.\n", iUserID );
    }
  }

  // ---------------------------------------------------------------
  // ディレクトリパスデータ入力ポートに、新しいデータが入力された場合
  // ---------------------------------------------------------------
  if( m_ParamDirIn.isNew() ) {
    // ディレクトリパスデータから、データを読み込む
    m_ParamDirIn.read();

    // 学習・推論オブジェクトに、ディレクトリパスを設定する
    // ( = 学習・推論オブジェクトの設定を変更する )
    if( m_cBD.SetPath( m_ParamDir.data ) ) {

      // 学習・推論オブジェクトに、ディレクトリパスが、一度以上設定されたことを示す
      bParam = true;

      fprintf( stdout, "Param files <%s>loaded.\n", (char*)m_ParamDir.data );
    }
  }
```

```
  // ------------------------------------------------------------------
  // 特徴量項目数データ入力ポートに、新しいデータが入力された場合
  // ------------------------------------------------------------------
  if( m_DataCountIn.isNew() ) {
    // 特徴量項目数データ入力ポートから、データを読み込む
    m_DataCountIn.read();

    // 特徴量項目数のチェック
    if( iCount !=  -1 && iCount != m_DataCount.data ) {
      // 特徴量項目数は、途中で変更してはならない
      fprintf( stdout, "Input data size cannot be changed.\n" );
      return RTC::RTC_OK;
    }

    // 特徴量項目数を保持
    iCount = m_DataCount.data;
  }

  // ------------------------------------------------------------------
  // 特徴量項目数データ入力ポートに、新しいデータが入力された場合
  // ------------------------------------------------------------------
  if( m_DataAryIn.isNew() ) {
    // 特徴量データ入力ポートから、データを読み込む
    m_DataAryIn.read();

    // 特徴量項目数をチェック
    if( iCount != (int)m_DataAry.data.length() ) {
      fprintf( stdout, "Input data size is invalid.\n" );
      return RTC::RTC_OK;
    }

    if( iCount !=  -1 /* 特徴量項目数が、一度以上入力された? */ &&
        bTalker       /* 話者 ID データが、一度以上入力された? */ &&
        bParam        /* 学習・推論オブジェクトに、ディレクトリパスが、一度以上設定された?
*/ ) {

      // 学習・推論オブジェクトに、特徴量項目数を設定
      if( ! m_cBD.SetChannel( iCount ) )
        return RTC::RTC_OK;

      // 学習・推論オブジェクトに、特徴量を渡して、学習・推論処理を行う
      if( ! m_cBD.DriveRna( iUserID, &(m_DataAry.data[0]), &iRet, &dTrust ) )
        return RTC::RTC_OK;

      // 学習・推論結果を、データ出力ポートに設定
      m_ResultID.data = iRet;
      // 合致度を、データ出力ポートに設定
      m_Trusty.data = dTrust;

      // 学習・推論結果を、データ出力ポートから、出力する
      m_ResultIDOut.write();
      // 合致度を、データ出力ポートから、出力する
      m_TrustyOut.write();
    }
    else {
        printf( "Beta RNA not ready, set param or talker.\n" );
    }
  }
  return RTC::RTC_OK;
}
```

以上の実装で、学習・推論コンポーネントの実装が完了した。RT ミドルウェアの RT コンポーネント作成フレームワークを用いることで、既存のライブラリβ-RNA の RT コンポーネント化を容易に行うことができた。

５.３. サービスポートについて

RT ミドルウェア OpenRTM-aist-0.4.0 では、新たに、サービスポートが導入された。前章までででは、データポートを使用した学習・推論コンポーネントの開発について説明したが、本章では、サービスポートを使用した学習・推論コンポーネントの開発について説明する。

５.３.１. サービスポートで公開する関数インターフェースの決定

サービスポートに対応した実装を行うことで、学習・推論コンポーネントの関数を、他の RT コンポーネントに提供できるようになる（＝他の RT コンポーネントが、直接、学習・推論コンポーネントの関数を呼び出せるようになる）。まずは、どのような関数を、外部の RT コンポーネントに公開するかを、決定した。

学習・推論コンポーネントが、サービスポートで公開する関数

関数名称	戻り値	戻り値の内容	引数		引数の内容
Init	short	処理結果	string	sRnaDir	パラメータディレクトリ
			short	iInputCh	特徴量の個数
Exec	short	処理結果	short	iTalkerID	話者 ID
			DArray(*)	aInputData	特徴量データ配列
			short&	iResultID	推論結果の話者 ID
			double&	dTrusty	確信度

(*) DArray：sequence<double>のこと。sequence は、サービスポートが提供する関数で使用可能な、配列テンプレートである。

なお、サービスポートで提供する関数の設計は、通常の関数設計と大きく異なる点はない。通常の関数設計と異なる点は、ポインタ渡しが行えない点のみである。しかしながら、参照渡しは可能であり、また、構造体及び、可変長配列も使用可能であるため、通常の用途であれば、不自由を感じることはない。ただし、多次元可変長配列を定義する事はできないので、この点には注意を要する。

５.３.２. 関数定義ファイルの作成

関数定義ファイルは、IDL（Interface Definition Language）形式で行う。IDL の書式については、CORBA（Common Object Request Broker Architecture、OMG が定めた分散オブジェクト技術の仕様である。RT ミドルウェアの基盤技術でもある。）の関連書籍等を参照すること。下記に、学習・推論コンポーネントで使用した、関数定義ファイルを示す。

```
interface BetaRna
{
  typedef sequence<double> DArray;

  short  Init( in  string   sRnaDir,
               in  short    iInputCh );
  short  Exec( in  short    iTalkerID,
               in  DArray   aInputData,
               out short    iResultID,
               out double   dTrusty );
};
```

sequence 型の変数は、可変長配列として扱うことができる。また、in で定義された引数は値渡し、out で定義された引数は参照渡しとして扱われる。

５.３.３. コンポーネント雛形の決定

サービスポート対応版の学習・推論コンポーネントの雛形を、コードジェネレータ rtc-template を使って、作成する。下に、サービスポート対応版の学習・推論コンポーネントの雛形を生成するために用いた、rtc-template 起動シェルスクリプトを示す。

```
#!/bin/bash
rtc-template  -bcxx \
   --module-name=BetaRnaService  --module-type='BetaRnaService' \
   --module-desc='BetaRna service port component' \
   --module-version=0.1  --module-category=Generic \
   --module-comp-type=COMMUTATIVE  --module-act-type=SPORADIC \
   --module-max-inst=10 \
   --service=BetaRna:BetaRna0:BetaRna \
   --service-idl=BetaRna.idl
```

[rtc-template のオプションについて]

サービスポートの指定方法

rtc-template のオプションで、サービスポートを定義する方法を説明する。関数を提供する側のコンポーネント（以後 provider）と、関数を使用する側のコンポーネント（以後 consumer）とでは、定義の書式が異なる。推論・学習コンポーネントは、関数を提供する側のコンポーネントなので、provider の書式を用いた。

provider での書式

サービスポートの指定	：	--consumer=サービス名 1:変数名:サービス名 2
関数定義ファイルの指定	：	--consumer -idl=関数定義ファイル（IDL ファイル）

上記の書式で雛形を生成すると、'サービス名１'を型としたサービスポート変数 m_'変数名'が、コンポーネントクラスのメンバ変数として定義される。通常は、'サービス名１'と'サービス名２'は、同一にしてよい。なお、consumer の rtc-template を実行するときは、provider の rtc-template オプションで指定したものと、同一の関数定義ファイルを、オプションで指定する必要がある。

５.３.４. 処理の記述

前章で示した rtc-template 起動シェルスクリプトを実行すると、以下のソースが生成される。

1) BetaRnaSVC_impl.h
2) BetaRnaSVC_impl.cpp

生成したソースのファイル名称及び、クラス名称は、rtc-template のオプションで指定した名称が使用される。

--consumer=サービス名 1:変数名:サービス名 2
==> ファイル名称：「'サービス名 2'SVC_impl.h」、「'サービス名 2'SVC_impl.cpp」
クラス名称　：「'サービス名 2'SVC_impl」

生成されたソースには、関数定義ファイルで定義した関数が、既に、作成されている。従って、上記 2)のソースファイルに作成された関数の中に、処理を記述するればよい。下に、サービスポート対応版の学習・推論コンポーネントの実装を示す。なお、m_cBD は、β-RNA ライブラリの、単純なラッパークラスである。

```
/*
 * Methods corresponding to IDL attributes and operations
 */
CORBA::Short BetaRnaSVC_impl::Init(const char* sRnaDir, CORBA::Short iInputCh)
{
  if( ! m_cBD.SetPath( (char*)sRnaDir ) )
    return 1;

  if( m_cBD.SetChannel( (int)iInputCh ) )
    return 2;

  return 0;
}

CORBA::Short BetaRnaSVC_impl::Exec
( CORBA::Short           iTalkerID,
  const BetaRna::DArray& aInputData,
  CORBA::Short&          iResultID,
  CORBA::Double&         dTrusty )
{
  int i, iCount, iRet;
  double dRet;
  double*pData;

  iCount = aInputData.length();
  pData = (double*)malloc( sizeof( double ) * iCount );
  if( ! pData )
    return 1;

  for( i = 0; i < iCount; i++ )
    pData[i] = aInputData[i];

  if( m_cBD.DriveRna( (int)iTalkerID, pData, &iRet, &dRet ) ) {
    iResultID = iRet;
    dTrusty = dRet;
    iRet = 0;
  }
  else {
    iRet = 2;
  }
  free( (void*)pData );
  return iRet;
}
```

実装に関しては、通常の C++ を理解していれば、キャストに注意する程度で十分であるが、関数の戻り値として char* を返す場合、以下のような記述にする必要がある。

```
Char*    retString = “ABC”;
return CORBA::string_dup( retString );
```

通常の C++の関数と異なり、サービスポート呼ばれた関数が、文字列を返す場合には、

RT ミドルウェアがソケット通信により、関数呼び出し側の RT コンポーネントへデータ転送を行った後に、文字列領域を開放しようとする。従って、retrun retString; とすると、文字列領域を開放できないため、エラーが生じることになる（ただし、動作上は、問題は無い）。

なお、rtc-template 起動シェルスクリプトの実行により、自動生成された「XXXX_impl」クラスに対して、何の実装も行わずに、make（make ファイルは、rtc-template 起動シェルスクリプトの実行により自動作成される）を行った場合は、その旨を、注意されることになる。

５.３.５. サービスポートを利用するコンポーネントの作成

本章では、サービスポート対応版の学習・推論コンポーネントを利用するコンポーネント（consumer）の作成方法を紹介する。

サービスポート対応版の学習・推論コンポーネントを利用するコンポーネントの雛形を、コードジェネレータ rtc-template を使い、作成する。下に、rtc-template 起動シェルスクリプトを示す。

```
#!/bin/bash
rtc-template -bcxx \
  --module-name=GUI --module-type='GUI' \
  --module-desc='Demo GUI component' \
  --module-version=0.1 --module-category=Generic \
  --module-comp-type=COMMUTATIVE --module-act-type=SPORADIC \
  --module-max-inst=10 \
  --inport=Size:TimedLong --inport=Wave:TimedShortSeq \
  --inport=DataCount:TimedShort --inport=DataAry:TimedDoubleSeq \
  --outport=Trigeger:TimedString --outport=File:TimedString \
  --outport=SendSize:TimedLong --outport=SendWave:TimedShortSeq \
  --consumer=TalkerMng:TalkerMng0:TalkerMng --consumer-idl=TalkerMng.idl \
  --consumer=PowerInfo:PowerInfo0:PowerInfo --consumer-idl=PowerInfo.idl \
  --consumer=BetaRna:BetaRna0:BetaRna --consumer-idl=BetaRna.idl
```

サービスポートの指定方法

rtc-template のオプションで、サービスポートを定義する方法を説明する。関数を使用する側のコンポーネント（consumer）は、以下の書式を用いる。

consumer での書式

サービスポートの指定	：	--service= サービス名 1:変数名:サービス名 2
関数定義ファイルの指定	：	--service-idl=関数定義ファイル（IDL ファイル）

上記の書式で雛形を生成すると、'サービス名１'を型としたサービスポート変数 m_'変数名'が、コンポーネントクラスのメンバ変数として定義される。通常は、'サービス名１'と'サービス名２'は、同一にしてよい。なお、関数定義ファイルは、サービス対応版の学習・推論コンポーネント開発時に作成したものを指定する。

サービスポートを使った関数の呼び出し方

サービスポート変数は、サービス対応版学習・推論コンポーネントへのアクセスポインタとなる。

```
RTC::CorbaConsumer<BetaRna> m_BetaRna0;
```

m_BetaRna0 の関数を呼び出す形式（通常のポインタ変数と同じ）で、サービス対応版学習・推論コンポーネントの提供する関数を利用することができる。

```
void CGuiCore::OnGetPower( void )
{
  int i, iRet;
  BetaRna::DArray          array;
  short                    iResult;
  double                   dTrusty;
  gchar                    sBuf[128];
  char*                    sName;

  // パワースペクトルを配列に代入
  array.length( m_pGUI->m_DataCount.data );
  for( i = 0; i < (int)array.length(); i++ )
    array[i] = m_pGUI->m_DataAry.data[i];

  // サービスポートで、学習・推論コンポーネントの学習・推論処理を呼び出す
  // (m_pGUI は、学習・推論コンポーネントを利用するコンポーネント)
  try {
    iRet = m_pGUI->m_BetaRna0->Exec( m_iTalker, array, iResult, dTrusty );
  }
  catch(...) {
    sprintf( sBuf, "Error in service port( BetaRna ), check connection." );
    Status( sBuf );
    return;
  }
  ... (略)...
```

なお、m_BetaRna0 の関数にアクセスする際は、例外処理の記述が必要である。これは、学習・推論コンポーネントが起動されていない、又は サービスポートが接続されていない状況で m_BetaRna0 の関数にアクセスした場合は例外を生じるからである。

その他・注意点

サービスポートで関数を提供する側において、サービス開始は、コンポーネント起動直後となる（サービスポートが未接続の場合、関数が呼び出されることが無いので、この場合はポート接続以後となる）。つまり、provider は、活性状態（Activate されている状態）であるか、不活性状態（Activate されていない状態）であるかに関係なく、サービスを提供することになる。従って、provider の初期化処理等を、onActivated 関数に記述してはならない（onActivated 関数が呼ばれるとは限らないため）。

５.４. GUI を持つコンポーネントを作成する場合

GUI を持つ RT コンポーネントを作成する場合、main 関数の修正が必要な場合が考えられる。なぜなら、通常の GUI ツールキットはメインループ関数がブロッキング動作するため、main() の最後にメインループを持って来る必要があるからである。

RT コンポーネントでは XXXComp.cpp に mai 関数を持ち、main 関数の最後に manager->runManager(); の行でブロッキング動作している。

これを manager->runManager(true); に変更しノンブロッキングとした後に GUI ツールキットのメインループ関数でブロッキングする様にすれば、最も簡単な修正で GUI を実現できる。

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

  // Initialize manager
  manager->init(argc, argv);

  // Set module initialization proceduer
  // This procedure will be invoked in activateManager() function.
  manager->setModuleInitProc(MyModuleInit);

  // Activate manager and register to naming service
  manager->activateManager();

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


  // GUI 追加部分
  g_cCore.Run();

  return 0;
}
```

５.５. コンポーネントのビルド

コードジェネレータ rtc-template の実行により、make ファイルが生成される。make を実行すると、コンポーネントの実効に必要なファイル（XXXX.so、XXXXComp）が、生成される。以下に、学習・推論コンポーネントの make の例を挙げる。

```
rm *.o
rm BetaRnaPort.so
rm BetaRnaPortComp

make -f Makefile.BetaRnaPort
```

この操作により、BetaRnaPort.so（学習・推論コンポーネントのライブラリファイル）と、BetaRnaComp（学習・推論コンポーネントを起動するための、実行ファイル）が生成された。

６．試験作業

６.１．コンポーネントの起動

RT コンポーネントを起動手順は、以下の通りである。

(1) ネームサーバを起動する。

(2) RT コンポーネントを起動して、ネームサーバに登録する。

これらの手順を実行するための、シェルスクリプトの例を、以下に示す。

```
#!/bin/sh
#

nsport='9876'                          ←― ネームサーバが使用するポート番号を指定する
hostname=`hostname`

term=`which kterm`

if test "x$term" = "x" ; then
    term=`which xterm`
fi

if test "x$term" = "x" ; then
    term=`which uxterm`
fi

if test "x$term" = "x" ; then
    term=`which gnome-terminal`
fi

if test "x$term" = "x" ; then
    echo "No terminal program (kterm/xterm/gnome-terminal) exists."
    exit
fi

rtm -naming $nsport                    ←― ネームサーバを起動する

echo 'corba.nameservers: '$hostname':'$nsport > ./rtc.conf
echo 'naming_formats: %n.rtc' >> ./rtc.conf
echo 'logger.log_level: TRACE' >> ./rtc.conf

$term  -e ./BetaRnaPortComp &          ←― RT コンポーネント"BetaRnaPort"を起動する
```

なお、RT コンポーネントは、起動時に、rtc.conf ファイルを参照して、ネームサーバに登録される。従って、rtc.conf ファイルで、ネームサーバを指定する必要がある。以下のように、rtc.conf ファイルを作成して、シェルスクリプトと同一階層に配置する。

```
corba.nameservers: localhost.localdomain:9876
naming_formats: %n.rtc
logger.log_level: TRACE
```

ネームサーバアドレス　　ネームサーバポート番号

６.２. RTCLink を用いた動作確認

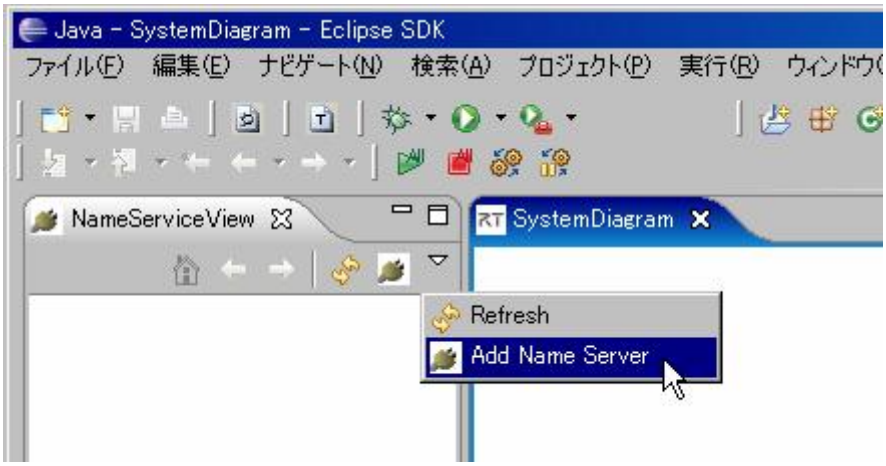
(1) RTCLink 起動する。

RTCLink の起動、操作方法は、以下のページを参照のこと。

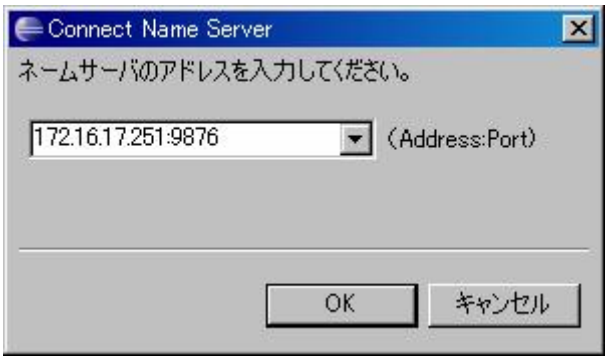
http://www.is.aist.go.jp/rt/OpenRTM-aist/html/FrontPage.html

(2) ネームサーバに接続する。

NameServiceView から、[AddNameServer]メニューを選択する（下図参照)。

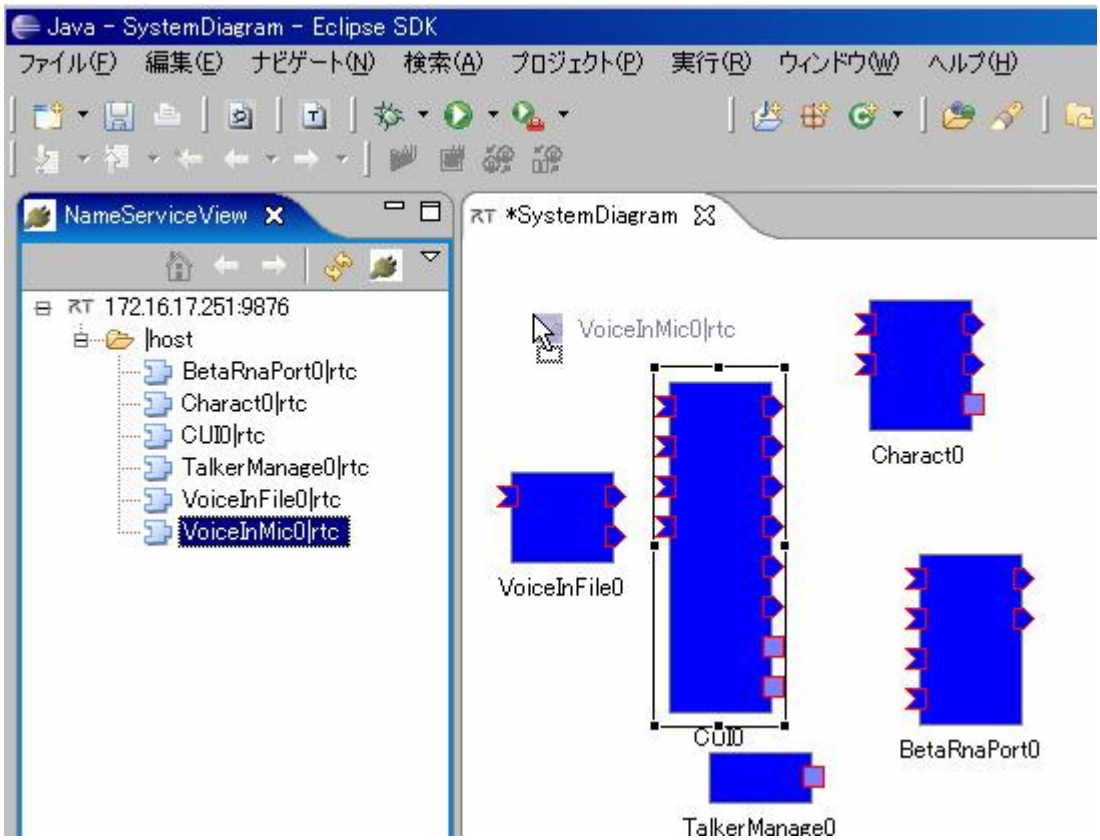


ConnectNameServer 画面で、ネームサーバを指定する（下図参照)。ネームサーバのアドレス、ポート番号は、前章で指定したものを指定する。

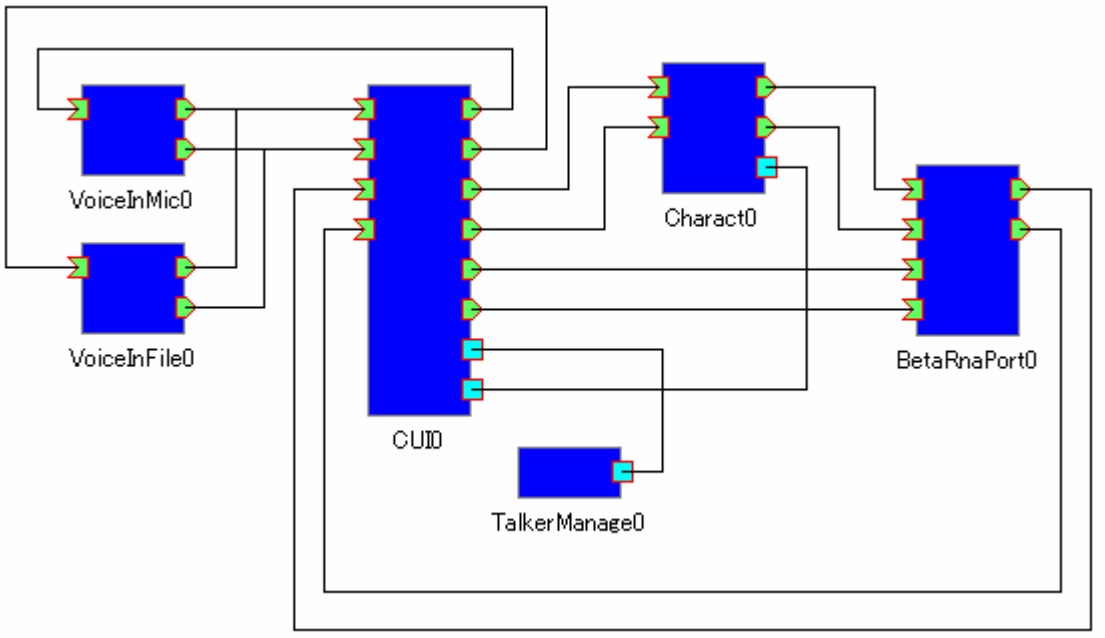


(3) RT コンポーネントの接続を行う

[ファイル]-[OpenNewSystemEditor]メニューを選択する。SystemDiagram ウインドウが作成されるので、そこに、NameServiceView で表示されている RT コンポーネントを、ドラッグアンドドロップ操作により、配置する。



次に、配置した RT コンポーネントのポート同士を接続する。



次に、ツールバーの All Active ボタン（下図参照）を押して、全ての RT コンポーネントをアクティブ状態にする（アクティブ状態になると、RT コンポーネントの色が、緑色（デ

フォルトの色設定）に変わる）。

以上の操作で、RT コンポーネントを使ったシステムが動作する。ここで、必要な確認を行う。なお、RTCLink 自体を終了しても、RT コンポーネント同士の接続が解除されたり、RT コンポーネントが終了したりするわけではないので、システムの動作は継続される。

６.３. コンポーネントの終了

RT コンポーネントの終了は、kill コマンドを使って行う。以下に、RT コンポーネントと、ネームサーバを終了するシェルスクリプトの例を示す。

```
#!/bin/sh

rnaportpid=`ps -C "BetaRnaPortComp" -o pid=`
if test -n "$rnaportpid"; then
    kill $rnaportpid
    echo "BetaRnaPortComp stopped."
fi

omnipid=`ps -C "omniNames" -o pid=`
if test -n "$omnipid"; then
    kill $omnipid
    echo "Naming service stopped."
fi
```

－以上－