3.1　ロボット知能ソフトウェアプラットフォームの開発

3.1.1 ロボット知能ソフトウェアプラットフォーム開発

次世代ロボットシステムの効率的かつ効果的な研究開発環境を構築するために、ロボット知能化技術をＲＴコンポーネントとしてモジュール化し、これらを統合して次世代ロボットシステムのシステム設計、シミュレーション、動作生成、シナリオ生成を行うことができるロボット知能ソフトウェアプラットフォームの研究開発を行った。RT コンポーネントのモジュール化に基づくシステム構築では、下図に示すようにロボットシステムを設計・開発するフェーズとそのロボットの動作を設計・開発する 2 つのフェーズが存在する。これらの開発を効率よく、効果的に開発を進めるためには、それぞれの開発ツールを統合的かつ違和感なく利用可能な開発環境が重要になる。

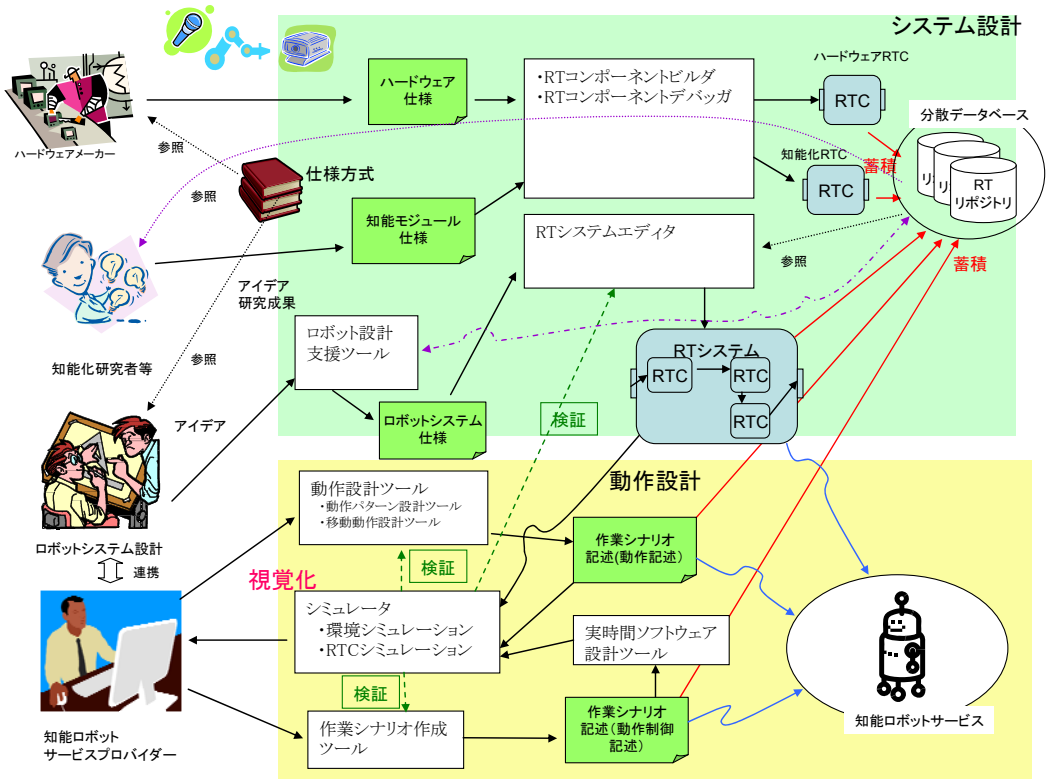

図 1　RT コンポーネントのモジュール化に基づくシステム構築の概要

そこで、本事業では、次世代ロボットシステムのシステム設計・開発を統合的に扱うことができるロボット知能ソフトウェアプラットフォームを実現するために、①ＲＴコンポーネント開発支援機能、②応用ソフトウェア支援機能及び③ロボットシステム設計支援機能の研究開発を実施した。

① RT コンポーネント開発支援機能

(a) RT システムに関する仕様記述方式

ソフトウェアプラットフォームの各種ツール間で共通的に利用する仕様記述方式の策定を実施した。仕様記述方式の策定に当たり、本事業実施機関内のみならず、本プロジェクトの各研究項目実施機関とも協議の上、多様なロボットシステムの構築に耐えうる汎用的かつ柔軟な仕様を策定し、最終的には、本プロジェクトの各研究項目実施機関が構築するロボットシステムプラットフォーム、ツール群、知能モジュール群は、策定された仕様記述方式に準拠したものとすることを目指す。

(a-1) ロボットシステム仕様記述方式

RT コンポーネントに基づく RT システムの構成を記述するための、ロボットシステム仕様記述方式を策定する。現在、産総研版 RT ミドルウェアが提供するツール RtcLink では RT コンポーネントによるシステム構成を記述する XML フォーマットを規定している。この記述方式を拡張し、十分な表現力を有するシステム仕様記述方式を策定する。最終目標として、本仕様を精査し安定版のロボットシステム仕様記述方式を策定するとともに、仕様を本プロジェクト外部に対しても公開する。また、OMG などの標準化組織での国際標準化を目指す。

RTS Profile 概要

RT システムがどのような RT コンポーネントから構成され、それらがどのように接続されているかを記述する方式 :「ロボットシステム仕様記述方式」を、RT System Profile: RTS Profile として仕様を定めた。RTS Profile は OMG RTC 標準で定められている RTC モデル[1]に基づき、使用する RTC と RTC が持つポート間の結合情報および、各 RTC のコンフィギュレーション情報によりシステムを記述するモデルとした。

RTS Profile データは図 2 に示すような、システム開発に関わる様々な場面での利用が想定されている。

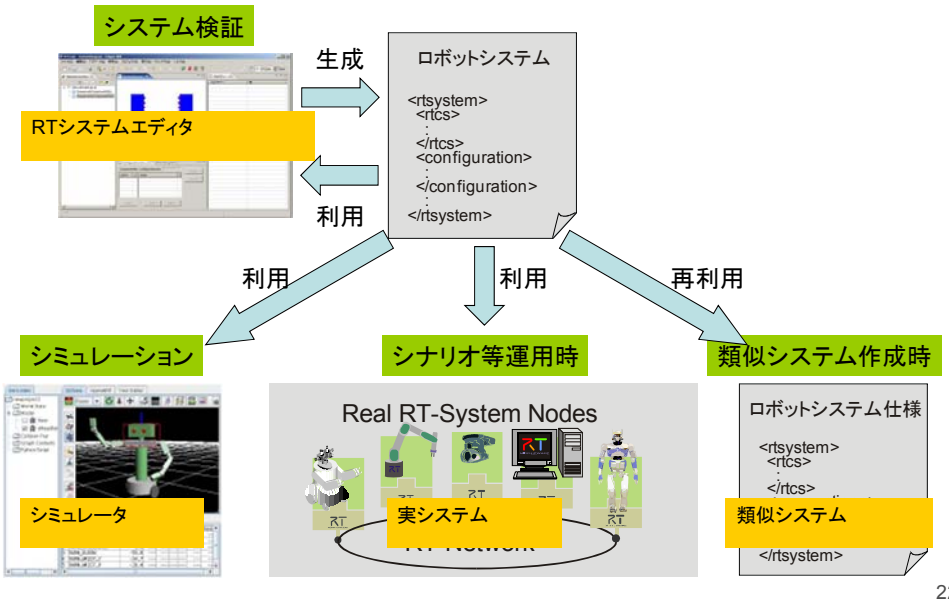

図 2 OpenRTP ツールチェーンにおける RTS Profile の利用例

システム構築・検証での利用

複数の RTC で構成される RT システムにおいては、システム構成情報はコンポーネントの配置情報・接続情報・コンフィギュレーション情報としてモデル化される。このシステム構成情報を記述するフォーマットとして RTS Profile を利用する。RT システム設計ツールである「RTSystemEditor」は、コンポーネントのプロファイル情報を読み込み、Eclipse 上でシステムの構成を編集、RTS Profile 形式の XML ファイルを生成することができる（図 3：オフラインエディタ機能)。

また、生成された RTS Profile の XML ファイルを読み込み、動作中の RTC の接続の再構成、コンフィギュレーション情報の設定、さらに動作を検証しつつ接続やコンフィギュレーション情報の再構築を行うことができる（図 4：オンラインエディタ機能)。

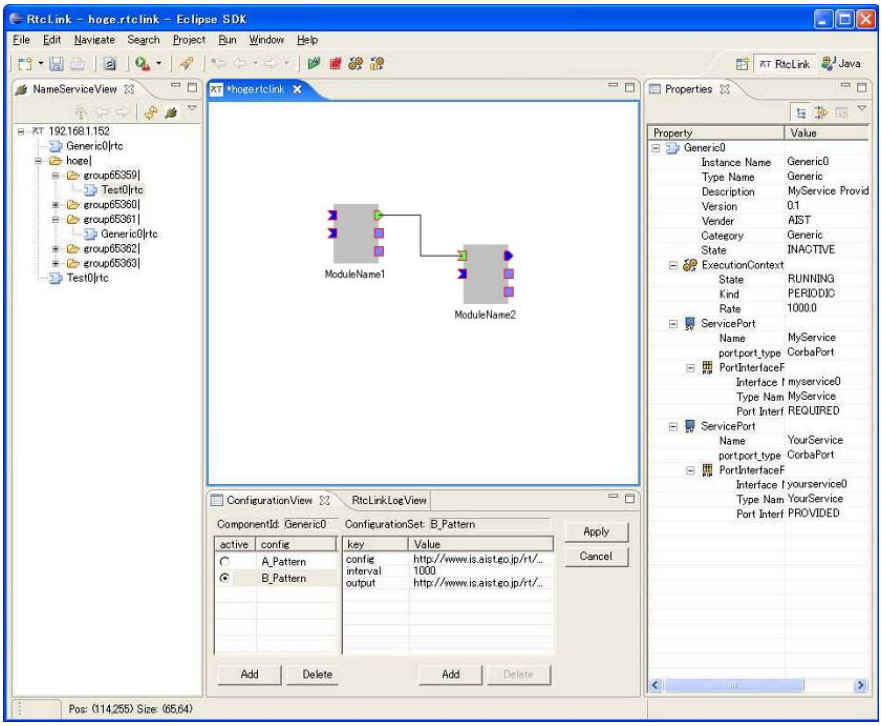

図 3 RT システムエディタ（オフライン編集画面）

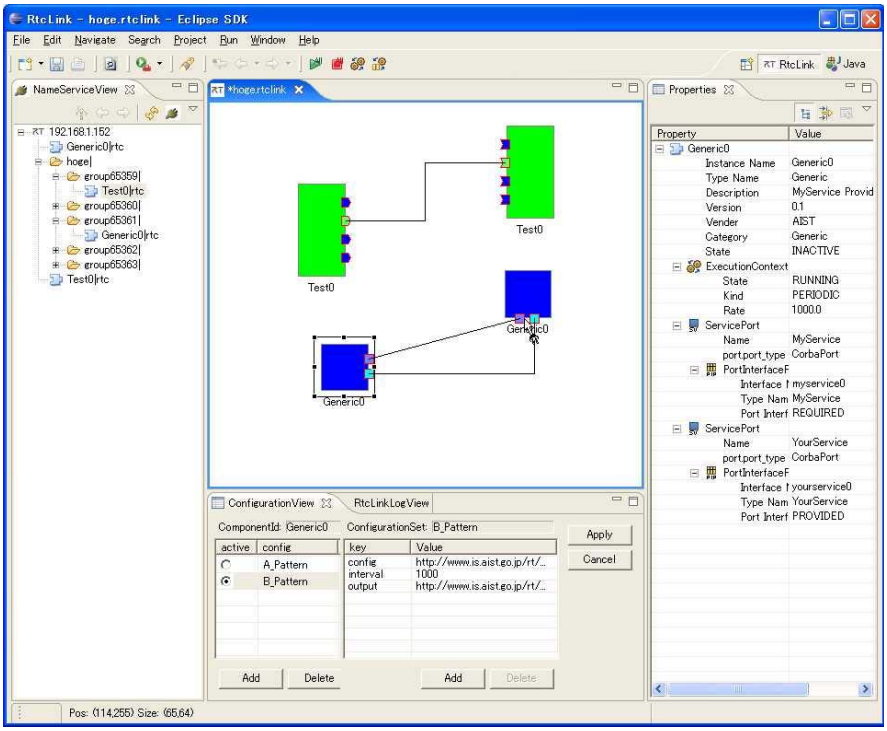

図 4 RT システムエディタ（オンライン編集画面）

システム運用時に利用

RTS Profile には、RT システムの RTC 間の接続情報、コンフィギュレーション情報などが含まれている。したがって、システム運用時にはこのファイルを元に、RTC の起動・接続・設定（Deployment and Configuration: D&C）を行うために利用することができる。

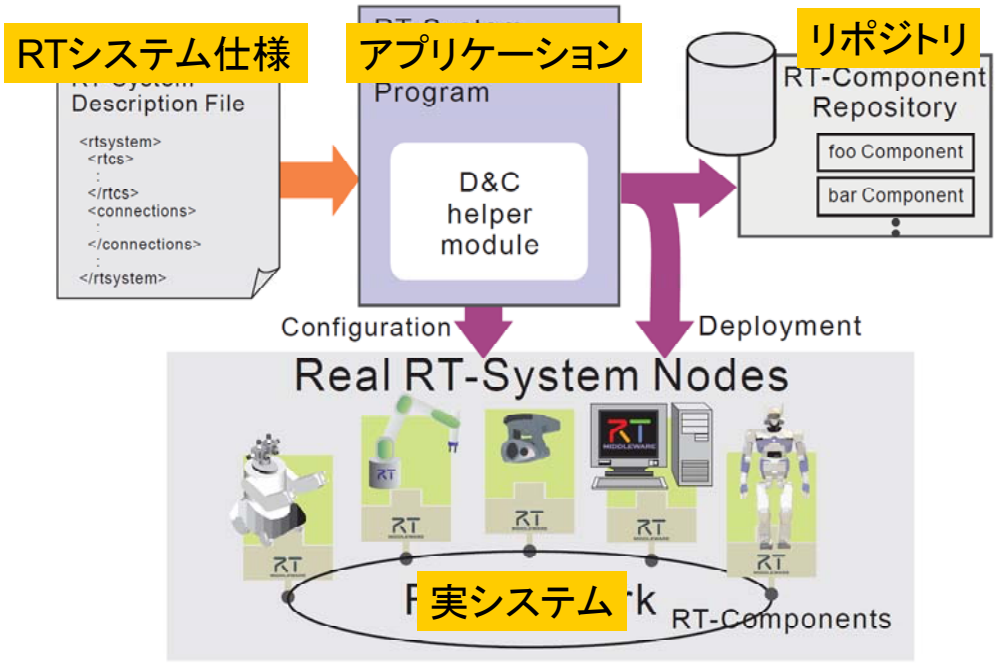

図 5 運用時のデプロイメントおよびコンフィギュレーション

図 5 にシステム運用時のイメージを示す。RTS Profile をアプリケーションが読み込み、仕様記述を元にアプリケーションに必要な RTC をリポジトリからノードに配置するとともに、RTC の接続・設定を行い、システム全体を起動する。

システム構成を動的に変更する場合には、個別の RTC を操作して構成を変更することも可能であるが、予め用意された複数の RTS Profile を切り替えることにより、迅速かつ容易にシステム構成を変更することができる。

また、シミュレータを利用する場合でも、実機の部分がシミュレータのコンポーネントに置き換わるだけで、基本的には同様の流れでシステムのデプロイおよび設定を行うことができる。

既存仕様の再利用

すでに存在するシステムと類似のシステムを設計する際には、既存の RTS 仕様を参照・再利用することで、システム構築にかかるコストを削減することができる。

RTS Profile 仕様記述方式

以上、RTS Profile の利用例を示したが、上述の用途以外にも仕様の明確なシステム構造を記述したファイルが存在すれば、静的・動的検証を含めたシステム安全検証、トレーサビリティの確保、動的デプロイメント・コンフィギュレーション等、様々な利用方法が考えられる。RTS Profile を各ツール間で利用するためには、その仕様を明確に定め仕様を安定的に維持する必要があり、その仕様記述方式を定めるにあたり MDA を採用した。

MDA、PIM および PSM

MDA（Model Driven Architecture）はソフトウェア標準化団体 OMG（Object Management Group）が提唱する、モデリング主導のシステムの開発、ライフサイクルの管理を実現するための参照アーキテクチャである[2]。中心となるモデルは、プラットフォームに非依存な PIM（Platform Independent Model）と プラットフォーム依存モデル PSM（Platform Specific Model）の 2 階層から構成される。PIM はプラットフォームに依存しないシステムのモデルであり、UML により特定の言語や OS、ミドルウェアなどに依存せず、かつ曖昧さを排除して構築されたモデルを指す。PSM は PIM から生成される各プラットフォームに特化したモデルであり、開発者はこのモデルを元に実装を行う。

モデルを実装依存部分と非依存部分に分け定義することで、技術トレンドが変化しても、PIM から PSM へのマッピングを定義し直すだけで、新たな技術に対応できる利点がある。

RTS Profile プラットフォーム非依存モデル

RTS Profile は RTS Basic Profile、RTS Extended Profile の 2 つのパッケージから構成される。図 6 に、RTS Profile のパッケージ構成を示す。

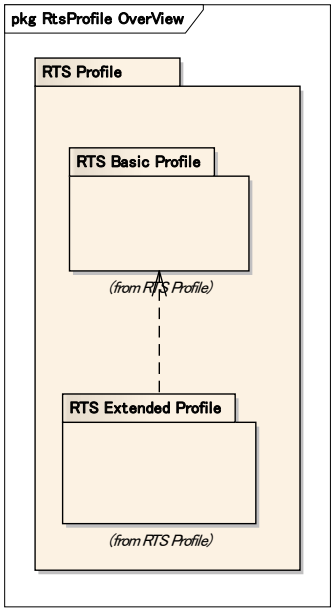

図 6 RTS Profile パッケージダイアグラム

それぞれのパッケージの主な役割は以下のとおりである。

- RTS 基本プロファイル (RTS Basic Profile)
- RTS 基本プロファイルは、RT システムの基本メタ情報を含む。
- RTS 拡張プロファイル (RTS Extended Profile)
- RTS 拡張プロファイルは、RT システムの本質的な機能にかかわらない付加的な情報や各種ツールでの利用を想定した情報を記述するために用意されているプロファイルである。

RTS プロファイルの PIM モデル全体図を図 7 に示す。

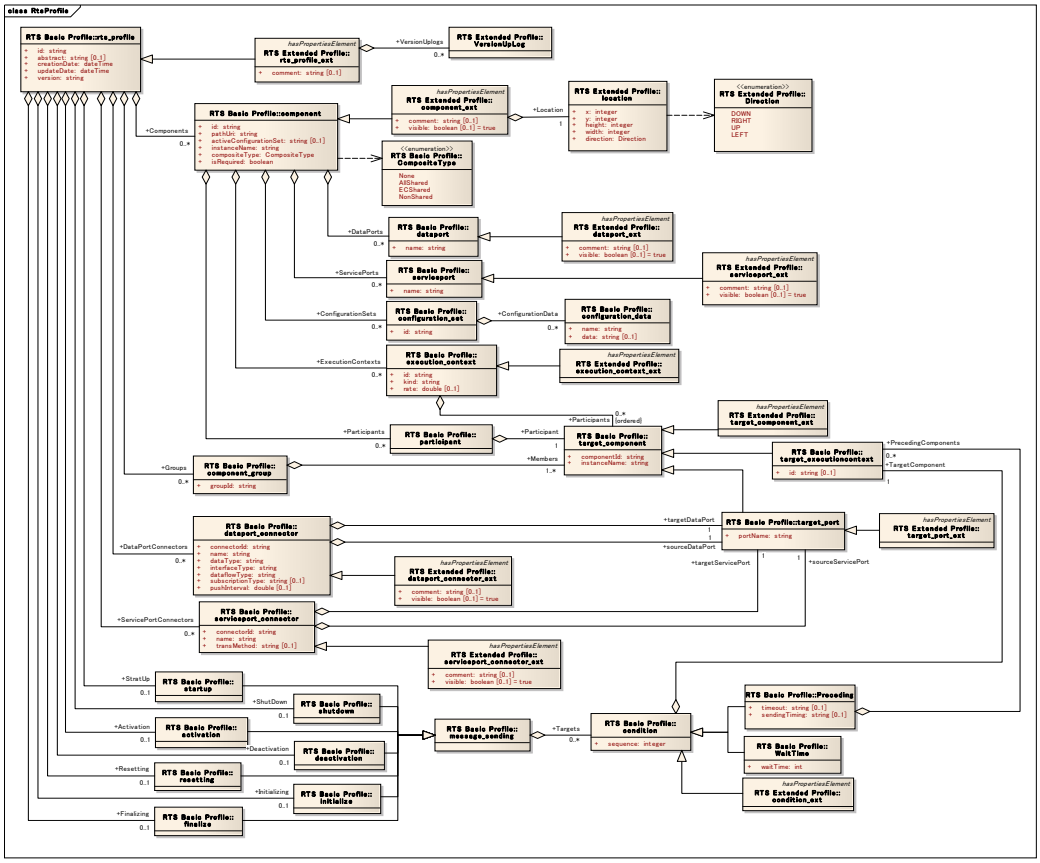

図 7 RTS Profile クラスダイアグラム（全体）

全 RTS Profile のうち、ここでは最も基礎となる rts_profile クラスについて概説する。

rts_profile は、RT システム仕様記述のルート要素である。また、仕様記述対象の RT システムを識別する ID 情報や RTC 仕様記述のバージョン情報を保持する。RTS プロファイル(PIM)の基本部分の拡大図を図 8 に示す。

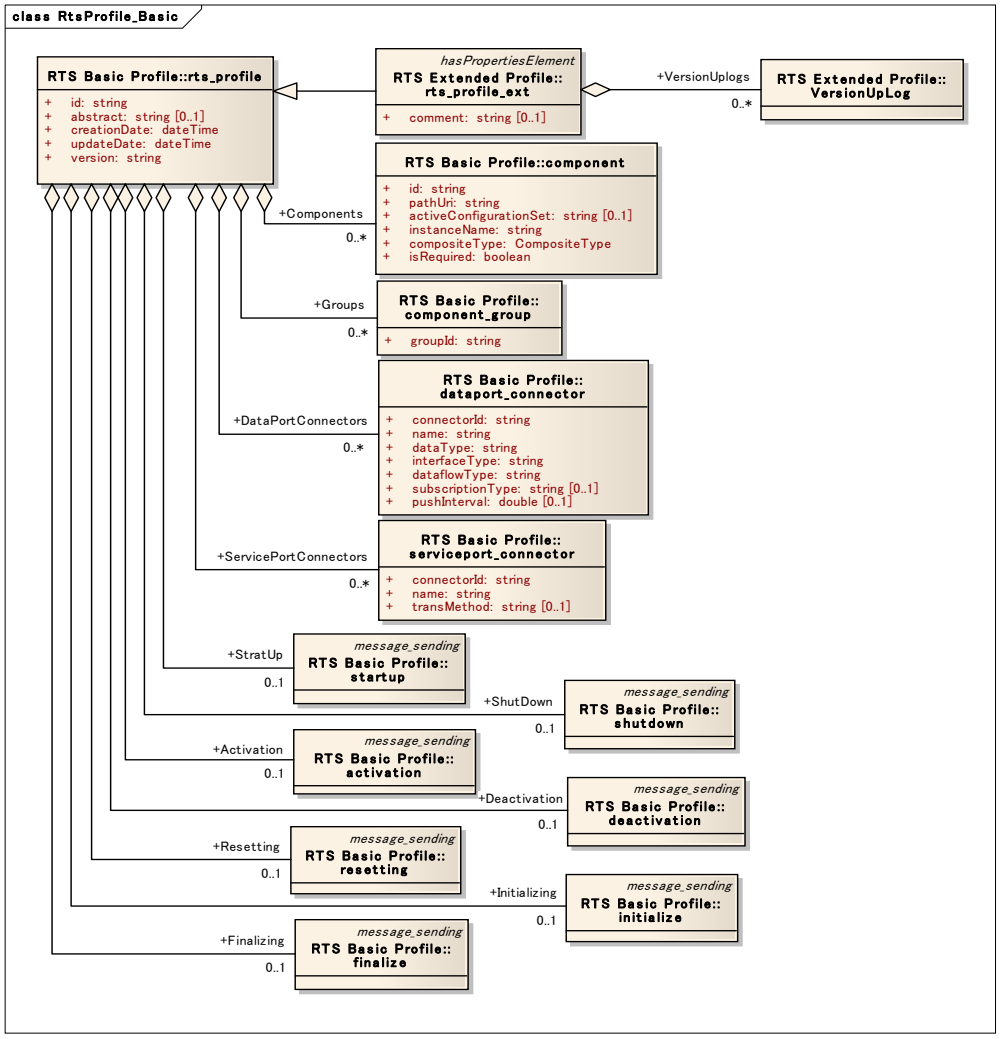

図 8 rts_profile クラスおよび関連部分

RT システム仕様記述の各種詳細情報は、rts_profile 要素以下の各要素に記述される（表 1)。

表 1 rts_profile 属性および関連

rts_profile		
attributes		
id	1	string
abstract	0..1	string
creationDate	1	dateTime
updateDate	1	dateTime
version	1	string
relationships		
Components	0..*	component

Groups	0..*	component group
DataPortConnectors	0..*	dataport connector
ServicePortConnectors	0..*	Serviceport connector
StartUp	0..1	startup
ShutDown	0..1	shutdown
Activation	0..1	activation
Deactivation	0..1	deactivation
Resetting	0..1	resetting
Initializing	0..1	initialize
Finalizing	0..1	finalize

- id : 仕様記述対象の RT システムを一意に識別するための識別子を指定する。識別子の書式は、以下の構成とする。
- RTSystem : [ベンダ名]. [システム名] : [バージョン番号]
- abstract : この属性は RT システムに関する説明を指定する。省略可能属性。
- creationDate : この属性は当該 RT システム仕様の作成日時を指定する。必須属性。
- updateDate : この属性は当該 RT システム仕様の最終更新日時を指定する。必須属性。
- version : RT システム仕様記述自体のバージョン番号を指定する。

この他のクラス定義については、添付資料 : RT システム仕様記述方式[3]を参照されたい。

プラットフォーム依存モデル

相互運用性を保証するために RTS Profile 仕様書では、XML (eXtensible Markup Language) および YAML (YAML Ain't Markup Language)の 2 種類のプラットフォーム依存モデル (PSM: Platform Specific Model) を定義している。
XML へのマッピングにおいて、基本型は以下のようにマッピングされる。

- string → xsd:string
- double → xsd:double
- integer → xsd:integer
- dateTime → xsd:dateTime
- boolean → xsd:boolean

また、パッケージと名前空間のマッピングは以下の通りに定義されている。

- RTS Basic Profile → rts
- RTS Extended Profile → rtsExt

詳細については、PIM 同様に添付資料：RT システム仕様記述方式[3]を参照されたい。

まとめ

策定した RT システム仕様記述方式はプロジェクト参加組織に対して公開すると共に、Web ページ (http://www.openrtm.org)において一般にも公開し、意見・要望などを調査した。本仕様記述方式は、RTSystemEditor や動力学シミュレータなどで使用され、ツール間のデータ連携に寄与している。RTS Profile の国際標準化活動に関して詳細は、次節の知能モジュール仕様記述方式のところで述べるが、RTS Profile および後述の RTC Profile を含め、RT コンポーネントの動的デプロイメントとコンフィギュレーションに関する各種データモデルやサービスインターフェースを標準化する取り組みを平成 20 年 12 月から OMG の Robotics DTF (Domain Task Force)・Infrastructure WG (Working Group)において行ない、DDC4RTC（RT コンポーネントに関する動的配置・設定に関する標準）の標準仕様策定作業における一次提案仕様に本仕様記述方式を取り入れた。

したがって、安定版のロボットシステム仕様記述方式を策定することができ、仕様を本プロジェクト外部に対しても公開することができた。また、国際標準化組織である OMG で DD4RTC の一部として国際標準化がすすめられている。

参考文献

[1] Object Management Group, “Robotic Technology Component Specification Version 1.0”, formal/2008-04-04, http://www.omg.org/spec/RTC/

[2] “Model Driven Architecture: Applying MDA to Enterprise Computing”, David S. Frankel, John Wiley & Sons, ISBN 0-471-31920-1

[3] 産業技術総合研究所 知能システム研究部門,「RT システム仕様記述方式」version 0.2, http://www.openrtm.org/openrtm/ja/node/943

(a-2)知能モジュール仕様記述方式

知能モジュール仕様記述方式は、RT コンポーネント化されたモジュール（知能モジュール等）のメタ情報及びその他の情報を含むデータ構造のための記述方式である。最終目標として、安定版の知能モジュール仕様記述方式を策定するとともに、仕様を本プロジェクト外部に対しても公開する。また、OMG などの標準化組織での国際標準化を目指す。

RTC Profile 概要

大規模なシステム開発や、モジュールの再利用を想定したシステム開発では、モジュールの設計者・実装者と利用者が異なるケースを想定すべきである。したがって、実装者から利用者に対してモジュールの設計意図振る舞い、データ仕様・インタフェース仕様を正確に伝達出来なければならない。こうしたモジュールの設計情報であるプロファイル、ポート、コンフィギュレーションといった基本情報や、コンポーネントの振る舞いを記述できる知能モジュール仕様記述方式を定めた。これを RTC Profile と呼ぶ。

RTC Profile は、OMG RTC 標準で定められている RTC モデル[1] に基づき、RT コンポーネントのモデルを記述すると共に、形式的に記述できない設計者の意図や詳細な振る舞いに関する情報を記述するドキュメント記述および拡張記述が可能な形式とした。RTC Profile のユースケースとしては以下のものが考えられる。

RTC Profile の利用

RTC Profile 仕様記述方式で記述された仕様は、RT システム開発プラットフォーム OpenRTP を構成するツールチェーンにおいて、ツール間のデータ交換のための標準フォーマットとして利用することを想定した。図 9 に RTC 仕様ファイルを利用するツール群の相互関係を示す。

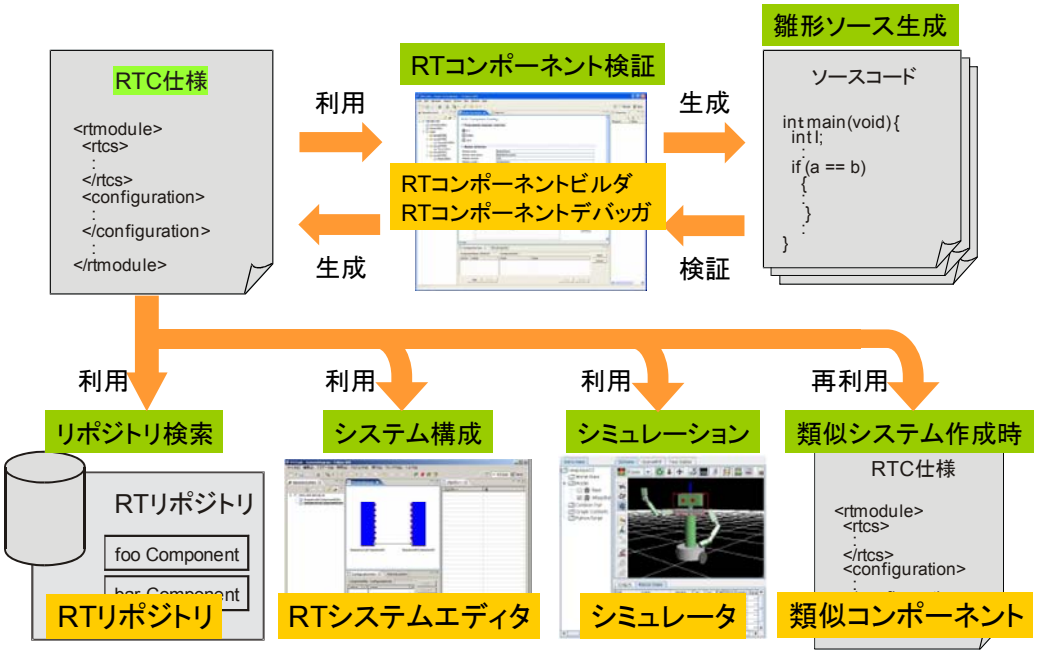

図 9 OpenRTP ツールチェーンにおける RTC 仕様の利用

RTC 仕様記述フォーマットとしての利用

システム設計時に、機能要素をモジュール分割し各要素の詳細設計を行う際の、各

モジュールの記述方式として利用する方法が考えられる。たとえば、UML でコンポーネントのモデルを記述し、その設計情報を XML 形式に変換し出力することで後述のコード生成の情報として利用する方法などが考えられる。
産総研が提供する RTC 設計ツール「RTCBuilder（図 10)」は Eclipse 上でモジュールの仕様を入力することで、XML 形式の RTC Profile を生成することができる。また、生成された RTC Porifle から作成された複数の実装の RTC 同士は、基本的な設計情報が同じであるため、コンポーネントレベルでは相互に互換性のあるものとなる。

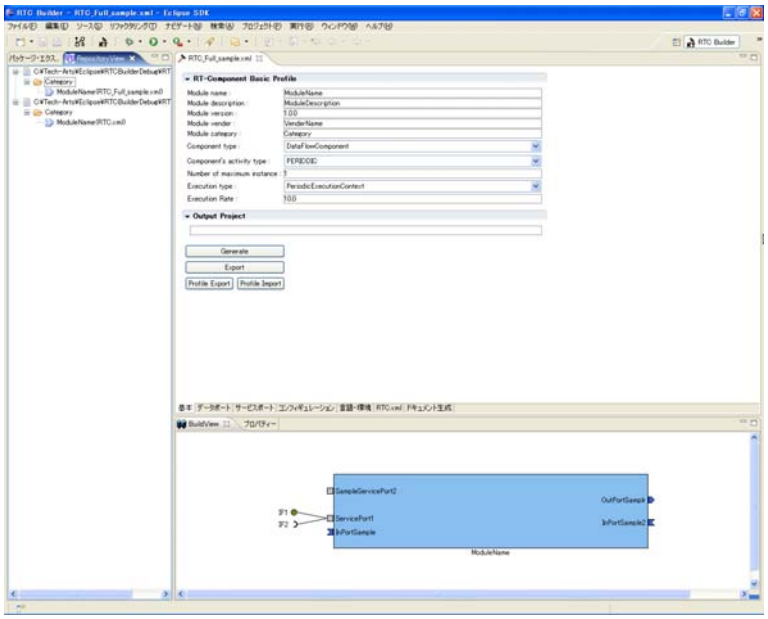

図 10 RTCBuilder の入力画面

コード生成のための情報として利用

RTC 仕様記述ファイルである RTC Profile には RTC のモデルに必要な情報がすべて含まれているため、この情報を元に RTC のひな型コードを生成することができる。新たな言語に対応した RT ミドルウェアが実装された場合でも、RTC モデルは共通であるので、この仕様記述方式に対応した新たなコードジェネレータを実装するだけで、既存の RTC 仕様から新言語に対応したコード生成を容易に行うことができる。本仕様記述方式は OMG RTC Specification[1] の PIM で定義されている要素をすべて含んでいる。したがって、OMG RTC Specification の PIM に基づき、CORBA 以外のプラットフォーム上に構築されたミドルウェアが実装された場合でも、本仕様記述方式で記述された RTC のモデルは有効であり、コードジェネレータなども容易に実装可能である。

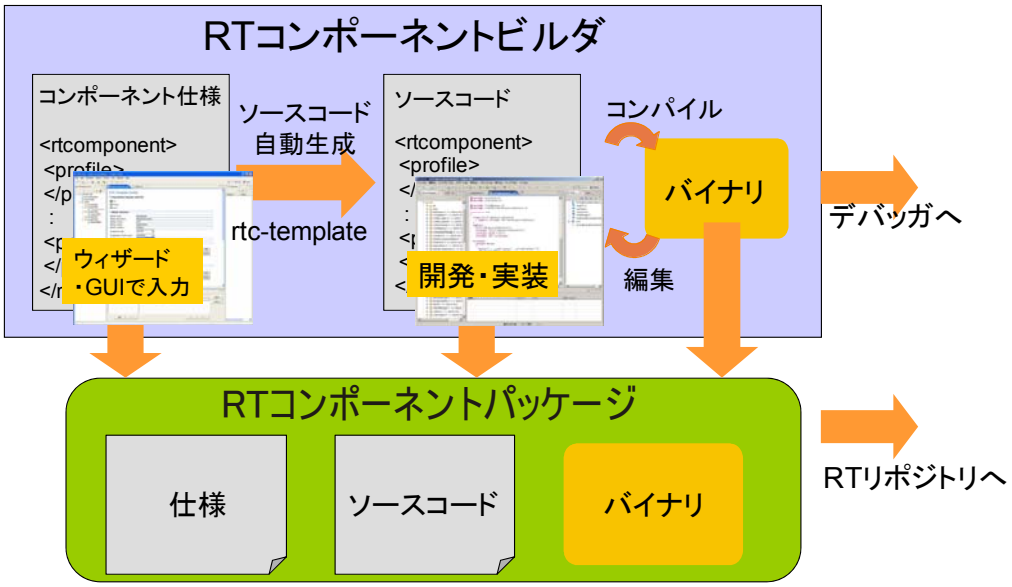

図 11 RTC ビルダによるコード生成と開発

産総研が提供する RTC ビルダでは、RTC の仕様入力による RTC 仕様記述ファイル（XML 形式）の生成と同時に、C++、Java、Python 用 OpenRTM-aist のコード生成を行うことができる（図 11）。

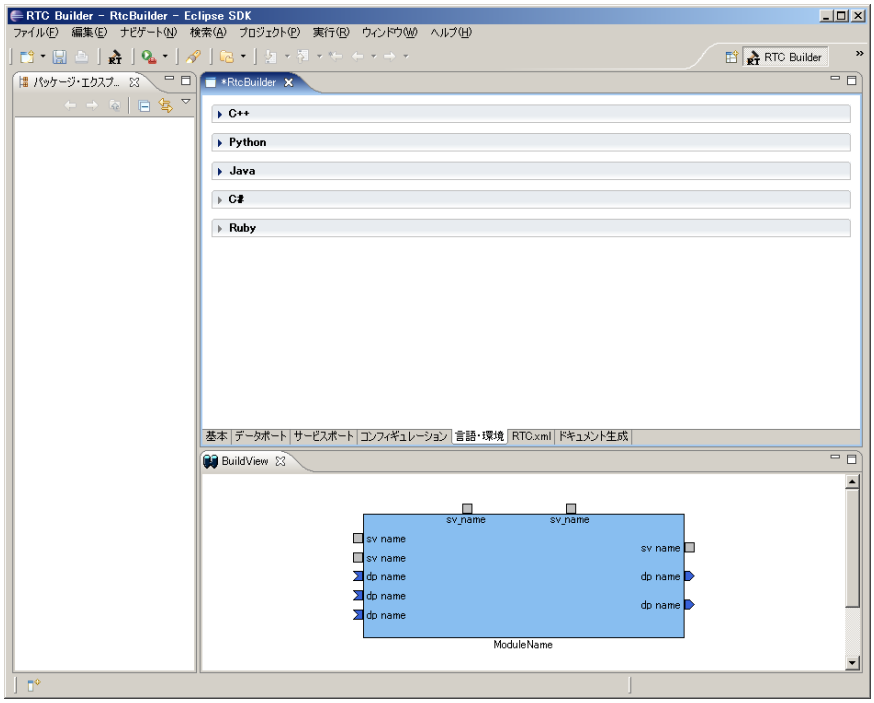

図 12 RTC ビルダの言語選択画面

RTC ビルダは、RTC 仕様記述に基づき、OpenRTM-aist の C++、Java および Python の RT コンポーネントのひな型コードを生成することができる（図 12）。作成されたコンポーネントは外部から見た際には同一のコンポーネントとして見える。コンポーネントを動作させる OS や利用可能な言語、実行速度、依存ライブラリなどの制約条件に合わせて実装言語を自由に選択することができる。

システム構成時に利用

RTC を実装する前の段階で、システム構成を行う作業に RTC Profile を利用することができる。図 13 は RTSystemEditor のオフラインエディタ画面である。オフラインエディタでは、実際にコンポーネントが起動していない状態でも、RTC 仕様を読み込みエディタ画面に RTC を表示、RTC を接続しシステムを構築することができる。RTC 仕様には、システム構築に必要な RTC プロファイルの情報が含まれているので、エディタには RTC のデータポートやサービスポートの数やそれぞれのプロファイル情報とともに、RTC をアイコンで表示し、ポートの接続や、コンフィギュレーション・パラメータの設定などの操作を行い、希望するシステム構成をオフラインで編集することが可能である。

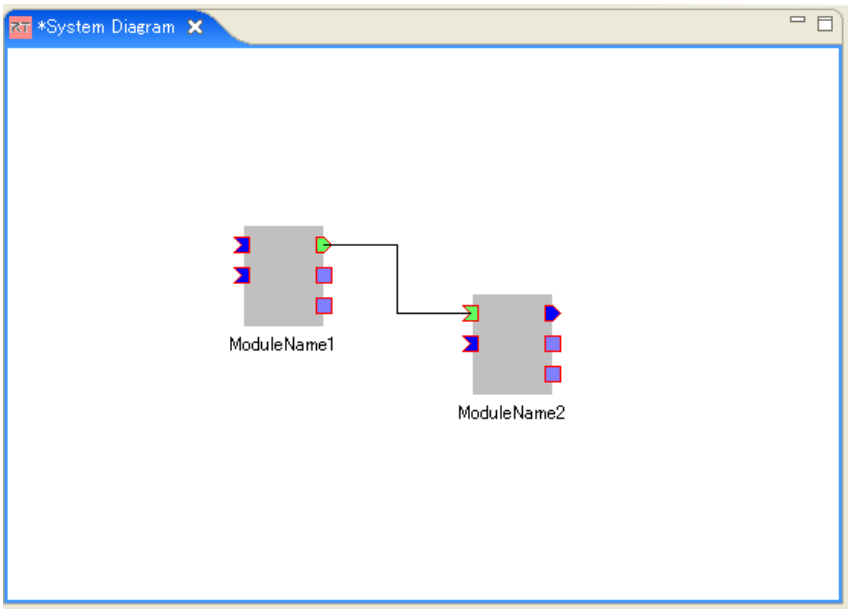

図 13 オフラインシステムエディタの編集画面

RTC リポジトリにおける検索情報として利用

RTC リポジトリは、RTC をサーバ上に蓄積し、必要な時に必要なコンポーネントを検索・ダウンロード・配置を行うための機能を提供するサーバである。RTC リポジトリにおいては、各コンポーネントは RTC の仕様、すなわち RTC Profile に基づいて登録・管理・分類することが可能である。

RTC Profile とともに登録されたコンポーネントは、利用者がコンポーネントを検索する際に入力する検索キーとマッチングを行い、リポジトリが適切なコンポーネントを利用者に提示する。
また、システム運用時には、コンポーネントの配置およびコンフィギュレーションを行うある種のアプリケーションプログラムが、システム仕様記述ファイルを解釈し、コンポーネントを選択、リポジトリサーバから適切なコンポーネントをダウンロード・配置・設定してシステムを起動することも可能である。このように、適切なコンポーネントを選択する際の情報として、RTC Profile を利用可能である。

既存仕様の再利用

すでに存在するコンポーネントと類似のコンポーネントを作成する際には、既存の RTC Profile を参照することで、より再利用性の高いコンポーネントを作成することができる。
例えば、ある種のセンサをコンポーネント化したものがすでにあり、そのデータポート、サービスポート、コンフィギュレーション・パラメータなどが再利用性の高い設計となっているとする。同種のより高性能なセンサが発売された場合、既存のセンサコンポーネントと同等の仕様でコンポーネントを作成することにより、すでにこのセンサコンポーネントが利用されているシステムにおいて、新たなセンサコンポーネントに置き換えることが容易にできるようになる。

ドキュメントとして利用

モジュールの再利用性を向上させるには、インタフェースの適切な定義と、十分なドキュメントを提供することが肝要である。モジュールの開発者と利用者が異なる場合や、モジュールのソースコードが提供されていない場合などを含め、モジュール利用者がモジュールの内部を詳細に調べなくても利用できるだけの十分な情報を提供することで、モジュールの再利用性は向上する。
RTC Profile にはプロファイル情報を記述する RTC Basic Profile とともに、RTC Basic Profile には記述しきれないセマンティックな情報を含めて RTC のドキュメント記述を支援する RTC Document Profile が定義されている。たとえば RTC Document Profile 内の doc_baisc::algorithm と呼ばれる要素には、そのコンポーネントが提供する機能やそれを実現するアルゴリズムを記述することで、利用者に対してモジュールがどういったアルゴリズムに基づき動作するかといった意図を伝える。また、doc_action::{description, precondition, postCondition}はそれぞれ、コンポーネントの Lifecycle State におけるそれぞれのアクションでどういった動作が行われるか、そのときの事前条件・事後条件といった意図を記述する。データポートやコンフィギュレーション・パラメータの Basic Profile ではそれらの名前や型に関

する情報が定義されるが、Document Profile ではそれらのデータの単位や意味といった情報を記述することができる。このように、RTC 仕様記述方式では、モジュール再利用のために利用者に対して必要かつ十分な情報を提供するための要素が数多く定義されている。

Document Profile を含む RTC 仕様の XML ファイルを、適当なテキストプロセッサに通すことで、人間が読みやすい形式に変換することも可能である。たとえば、RTC Profile の XML ファイルを、ドキュメント生成ツールである doxygen フォーマットに変換することで、HTML や LaTeX あるいは man 形式に変換しモジュール利用者に提供するなどの利用法も考えられる。図 14 に Doxygen により自動生成された RT コンポーネントのドキュメント例を示す。

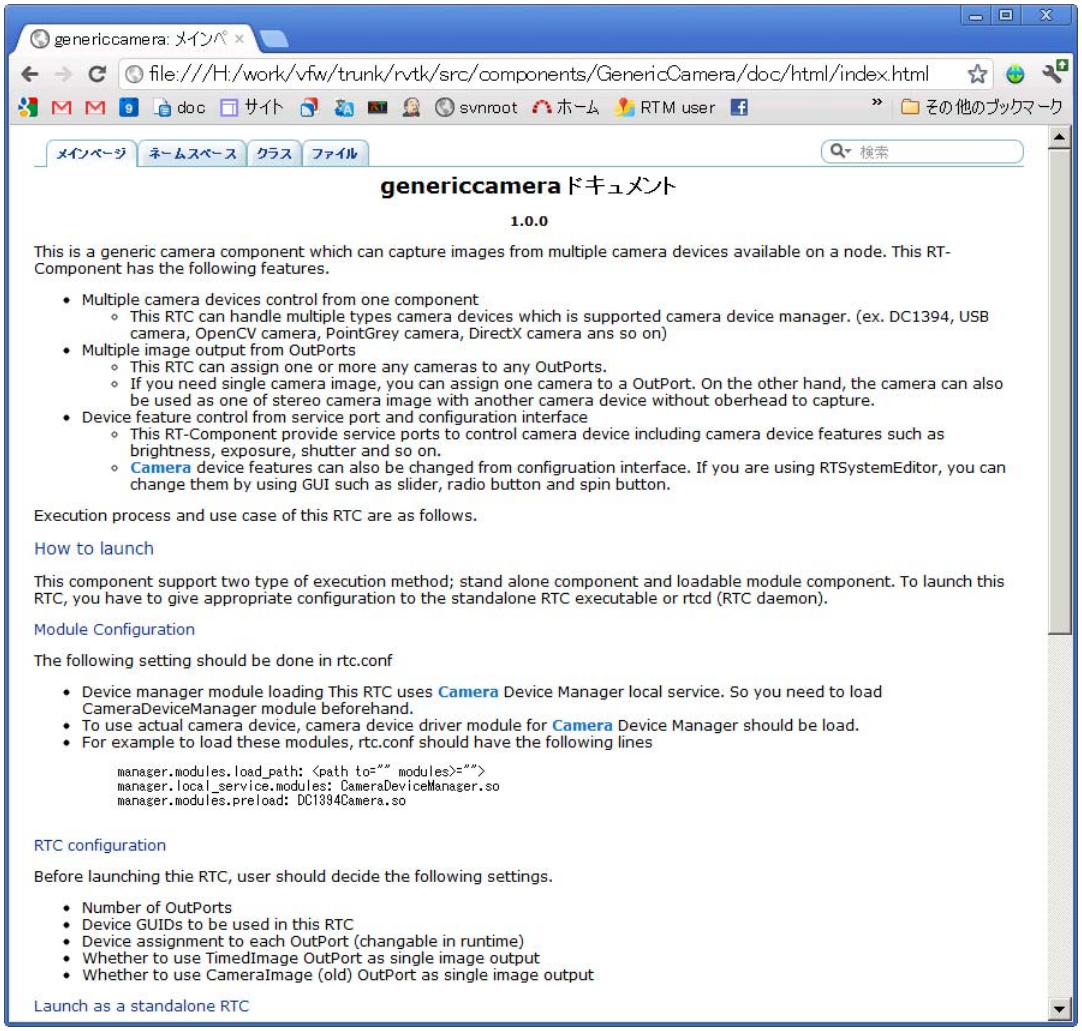

図 14 Doxygen により自動生成された RT コンポーネントのドキュメント

RTC Profile モデル

以上、RTC Profile の利用例を示したが、上述の用途以外にもコンポーネント仕様を記述したファイルが存在すれば、コンポーネントの静的・動的検証を含めたテストコードの自動生成、トレーサビリティの確保、パッケージの自動化等、様々な利用方法が考えられる。仕様の記述方式については上述の RTS Profile 同様、MDA を採用した。

RTC Profile プラットフォーム非依存モデル

プラットフォーム非依存モデル（Platform Independent Model：以後 PIM とする）は特定の言語や記述フォーマットに依存しないソフトウェアまたはデータのモデルであり、その実体は UML で記述されたモデル図と詳細を記述した文書から構成される。

PIM は以下の 3 つのパッケージから構成される:

- RTC Basic Profile. RTC の基本的なコンポーネントモデルを記述する際に必要な情報を定義する。RTC の基本プロファイルおよびポートやコンフィギュレーション情報がこれに含まれる。
- RTC Extended Profile. RTC Basic Profile に含まれないオプショナルな雑多な情報や各種ツールでの利用を想定した情報を記述するためのデータ構造を定義する。
- RTC Documentation Profile. 上記プロファイル情報以外に、詳細な仕様を文書として記述するためのデータ構造を定義する。

図 15 に、上記 3 つのパッケージの関係を示す。

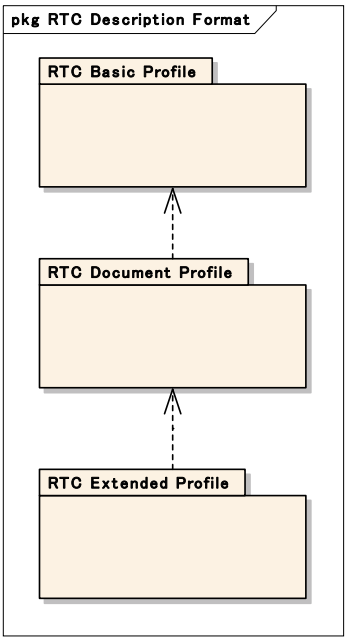

図 15 RTC Profile パッケージダイアグラム

RTC プロファイルは、ロボット用コンポーネントの各種属性を定義するデータモデルである。RTC プロファイルの PIM の全体図を図 16 に示す。また、プロパティ情報に関係するクラスの部分を図 17 に、各種制約情報を表現するための要素に関するクラス図を図 18 にそれぞれ示す。

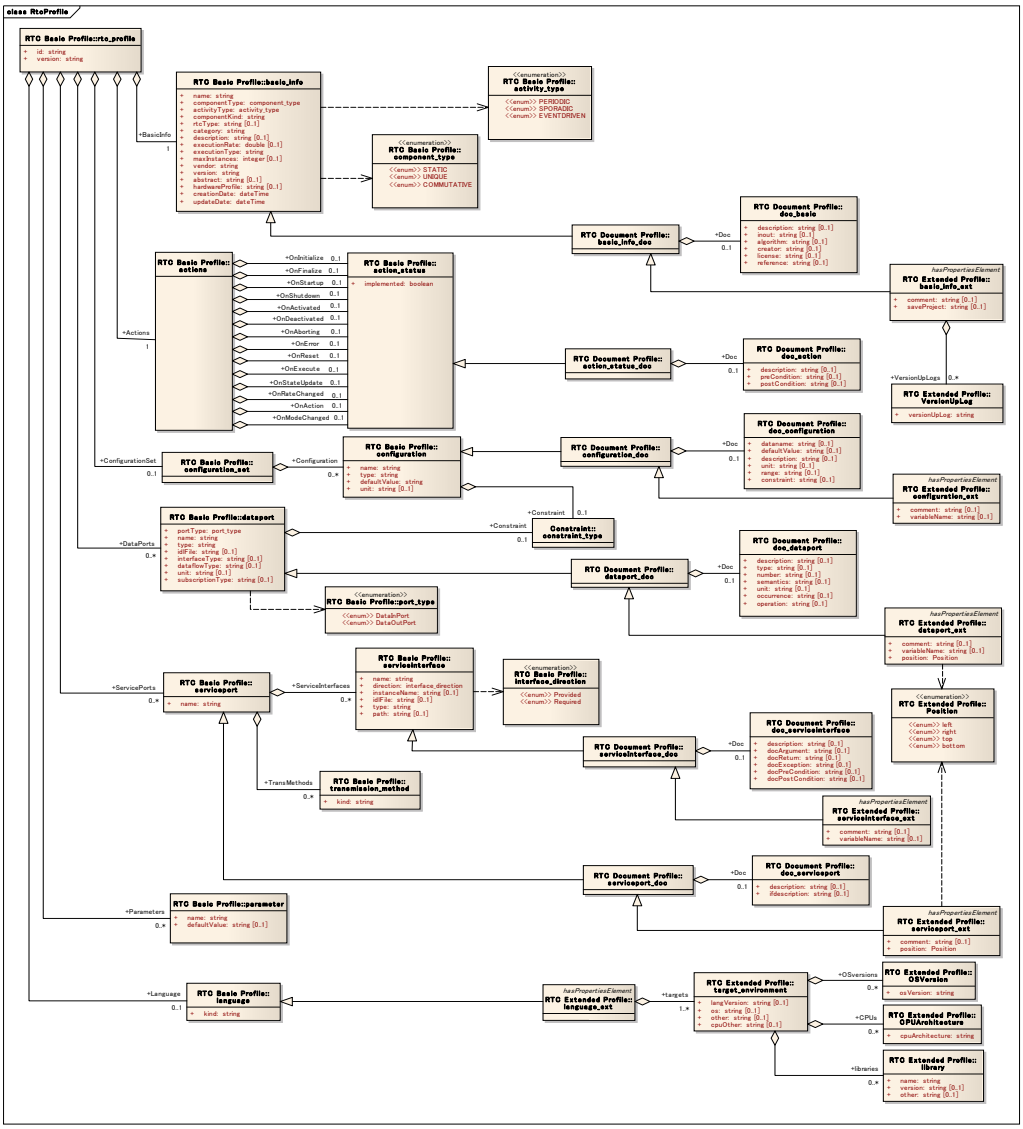

図 16 RTC Profile クラスダイアグラム（全体図）

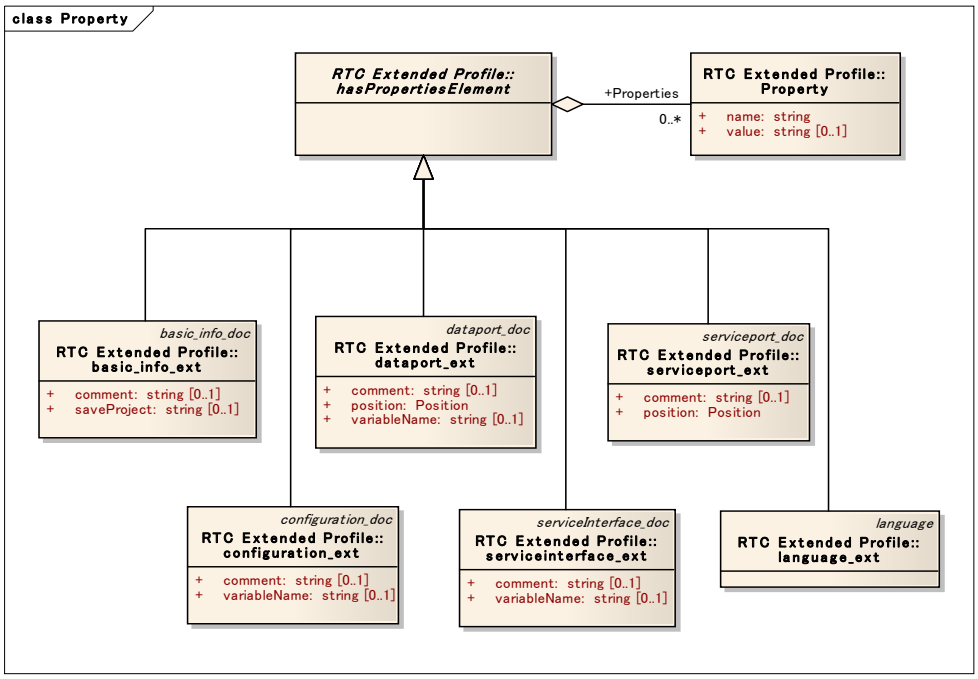

図 17 RTC Profile(プロパティ部分)

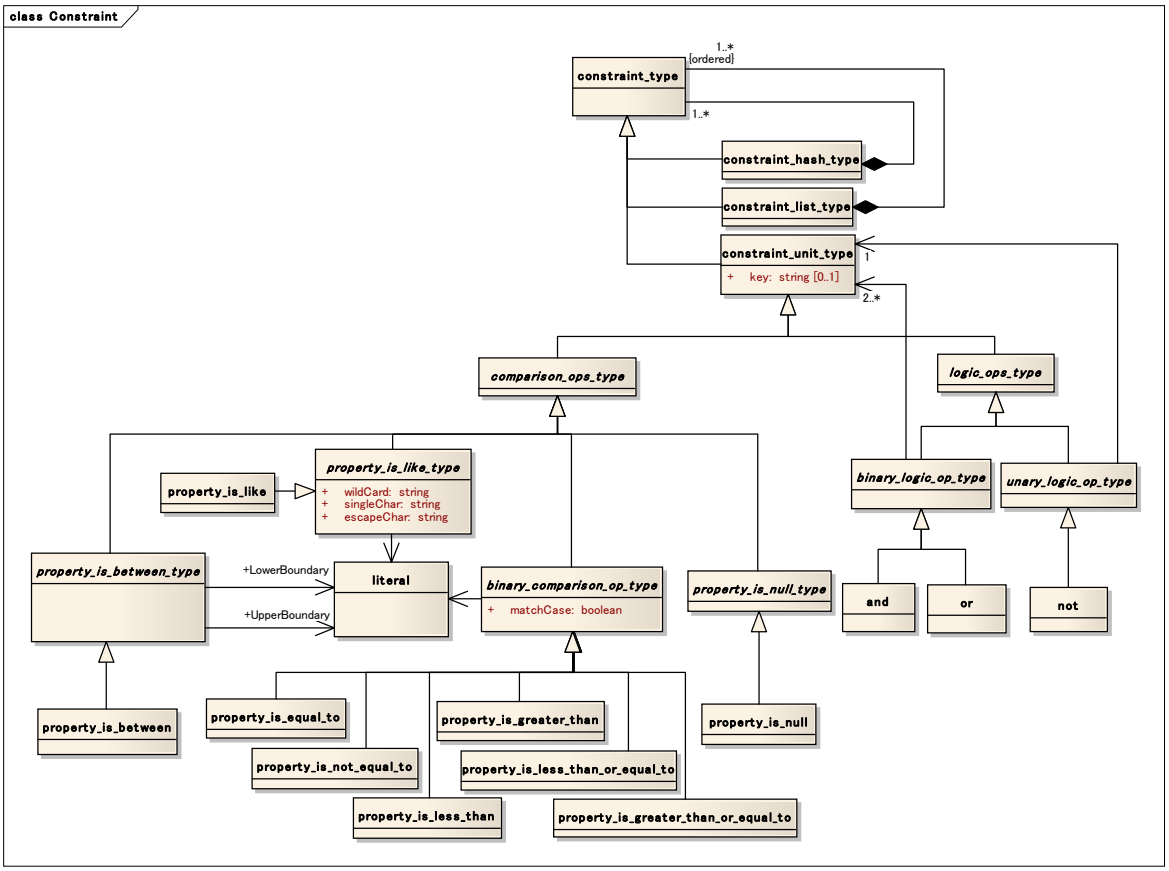

図 18 RTC Profile (制約部分)

全 RTC Profile のうち、ここでは最も中心となる rtc_profile クラスについて概説する。rtc_profile は、モジュール仕様記述のルート要素であり、仕様記述対象の RTC を識別する ID 情報や RTC 仕様記述のバージョン情報を保持する。rtc_profile (PIM) の基本部分の拡大図を図 19 に示す。

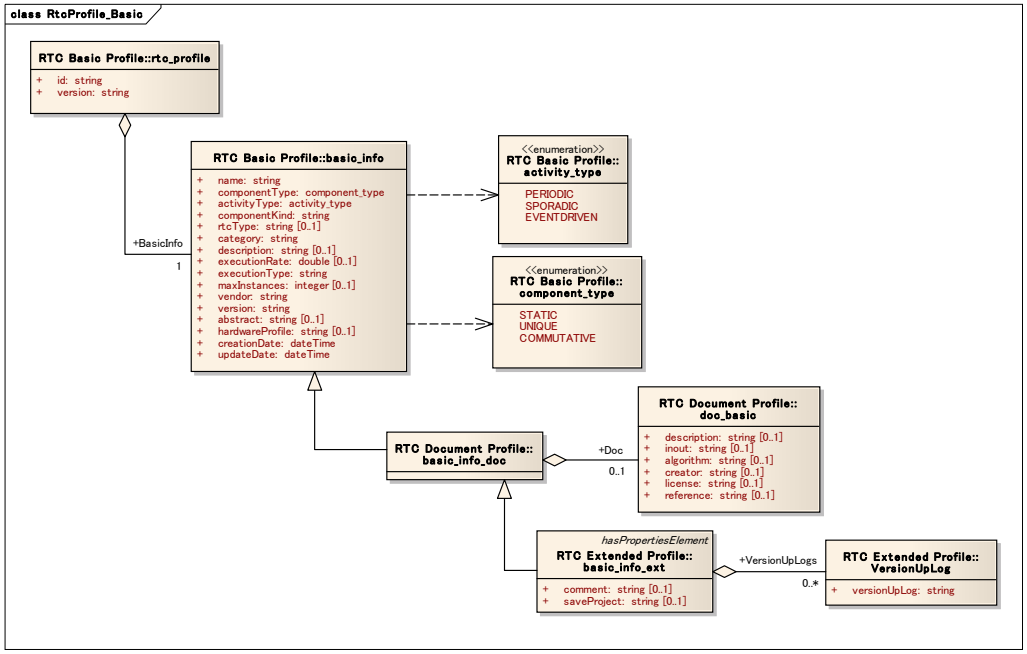

図 19 rtc_profile (コンポーネント記述)

rtc_profile は、RTC 仕様記述のルート要素である。また、仕様記述対象の RTC を識別する ID 情報や RTC 仕様記述のバージョン情報を保持する。RTC 仕様記述の各種詳細情報は、rtc_profile 要素以下の各要素に記述される。以下に rtc_profile の属性を示す（表 2)。

表 2 rtc_profile 属性および関連

rtc_profile		
attributes		
id	1	string
version	1	string
relationships		
BasicInfo	1	basic info
Actions	1	actions
ConfigurationSet	0..1	configuration set
DataPorts	0..*	dataport
ServicePorts	0..*	serviceport
Parameters	0..*	parameter
Language	0..1	language

- id：仕様記述対象の RTC を一意に識別するための識別子を指定する。識別子の書式は以下の構成とする。RTC : [ベンダ名] : [カテゴリ名] : [コン

ポーネント名] : [バージョン番号]

- version : RTC 仕様記述自体のバージョン番号を指定する。
- basic_info : basic_info は RTC の基本情報を記述する要素である。以下に basic_info の内容を示す（表 3)。

表 3 basic_info 属性

basic_info		
attributes		
name	1	string
componentType	1	component type
activityType	1	activity type
componentKind	1	string
rtcType	0..1	string
category	1	string
description	0..1	string
executionRate	0..1	double
executionType	1	string
maxInstances	0..1	integer
vendor	1	string
version	1	string
abstract	0..1	string
hardwareProfile	0..1	string
creationDate	1	dateTime
updateDate	1	dateTime
no relationships		

この他のクラス定義については、添付資料：RTC 仕様記述方式[2]を参照されたい。

プラットフォーム依存モデル

相互運用性を保証するために RTC Profile 仕様書では、RTS Profile 仕様書同様 XML (eXtensible Markup Language) および YAML (YAML Ain't Markup Language)の 2 種類のプラットフォーム依存モデル（PSM: Platform Specific Model）を定義している。

XML へのマッピングにおいて、基本型は以下のようにマッピングされる。

- string → xsd:string
- double → xsd:double
- integer → xsd:integer
- dateTime → xsd:dateTime
- boolean → xsd:boolean

また、パッケージと名前空間のマッピングは以下の通りに定義されている。

- RTC Basic Profile → rtc
- RTC Document Profile → rtcDoc
- RTC Extended Profile → rtsExt

詳細については、PIM 同様に添付資料：RTC 仕様記述方式[2]を参照されたい。

標準化活動

活動概要

上述の RTS Profile および RTC Profile を含め、RT コンポーネントの動的デプロイメントとコンフィギュレーションに関する各種データモデルやサービスインターフェースを標準化する取り組みを平成20年12月からOMGのRobotics DTF (Domain Task Force)・Infrastructure WG (Working Group)において行なってきた。同規格に関心を持つ韓国の ETRI（Electronics and Telecommunications Research Institute）とともに、平成 22 年 6 月（米国・ミネアポリスミーティング）に標準仕様公募文書：Dynamic Deployment and Configuration for RTC (DDC4RTC) RFP (Request for Proposal) を提出し、標準仕様の提案募集を開始した。同年 12 月（米国・サンタクララミーティング）に産総研、ETRI それぞれが DDC4RTC RFP に対する初期提案（Initial Submission）を行なった。その後、両提案を統合するための議論を重ね、平成 24 年 3 月（米国・ワシントンミーティング）の OMG 技術会議に提出された。MARS （Middleware And Related Systems）PTF (Platform task Force)に於いて原案は承認されたものの、標準作業部会 (AB: Architecture Board) において、事前提出からの修正点が多く、Board Member が十分に審議できる時間を確保できていないとして、次回平成 24 年 6 月（米国・ケンブリッジミーティング）に再提出するよう勧告された。Board Member からの指摘を修正し、次回再提出する予定である。次回の AB において承認されれば、最終文書化委員会（FTF : Finalization Task Force）を開始し、1 年間の標準仕様文書の整合性・実現可能性の検討を経た上で標準仕様として策定・一般公開される見込みである。

DDC4RTC 標準仕様概要

DDC4RTC 仕様は RTC の動的デプロイメント、すなわちコンポーネントのノードへの配置と設定を行うための標準規格である。OMG ではすでにコンポーネントの配置（デプロイメント）および設定（コンフィギュレーション）を行うための標準規格：DEPL（Deployment and Configuration of Component-Based Distributed Applications）[3] が策定されている。しかしながら、この標準では例えば図 20 のようなロボットシステムに特有なシステムの動的構成変更についてはサポートしておらず、想定しているコンポーネントモデルも RTC とは若干異なる。DDC4RTC

では DEPL に対して RTC に準拠したコンポーネントモデルをサポートするための若干の拡張と、動的システム構成変更のための仕組み（SupervisorFSM および ApplicationSupervisor）を付加した。

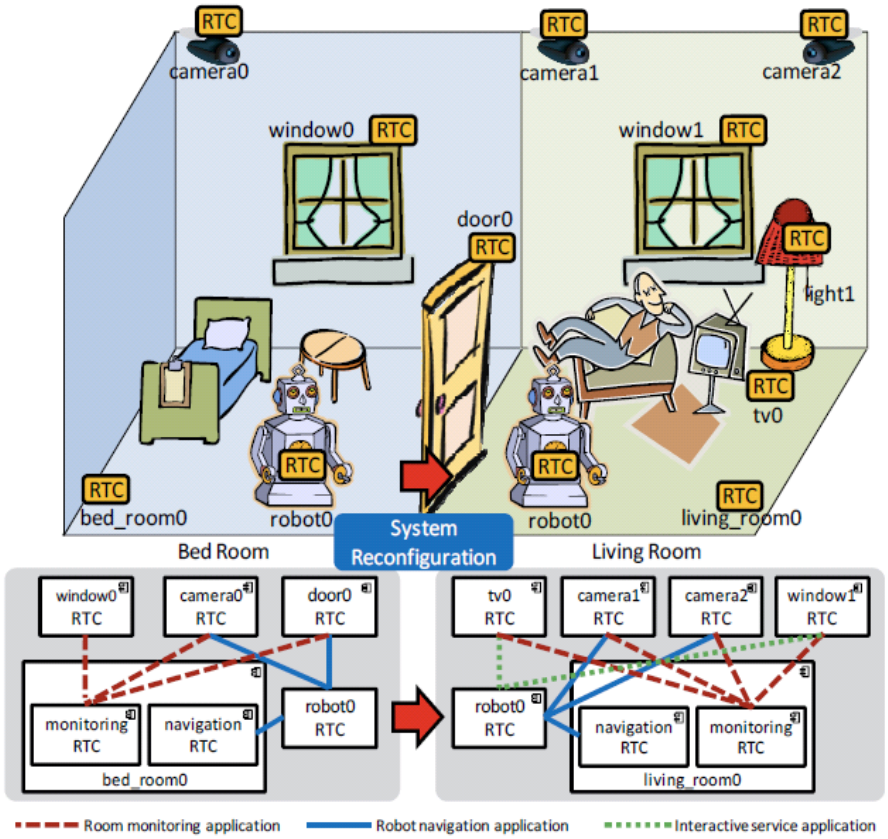

図 20 DDC4RTC が想定する動的システム

SupervisorFSM は多数のコンポーネントを監視しつつ、システム状態ごとに割り当てられた RTS Profile を各種イベントに応じて切り替え、システム構成の変更を行う。SupervisorFSM の基本的概念図を図 21 に示す。

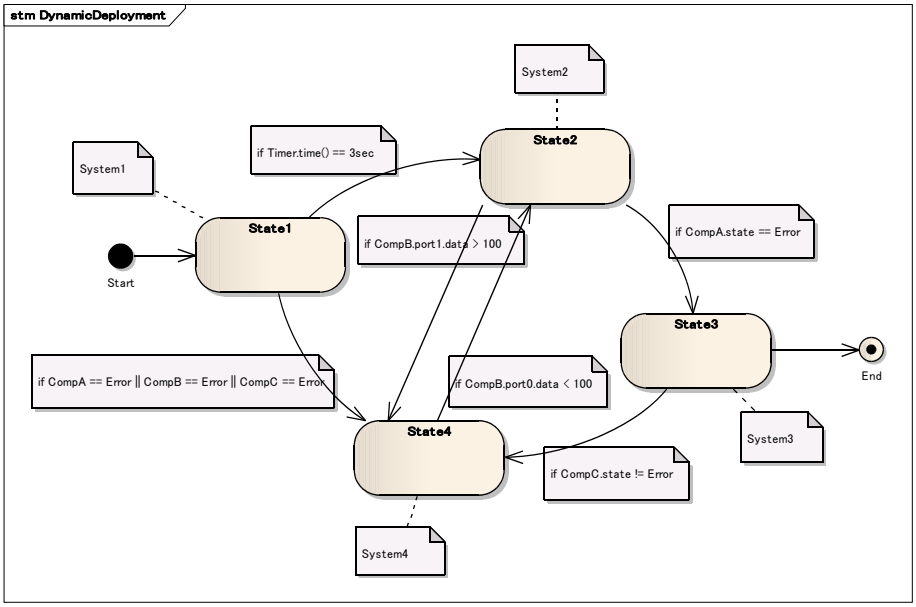

図 21 RT システムの動的構成変更の概念図

図 22 に DEPL 標準仕様と DDC4RTC 標準仕様の関係を示す。

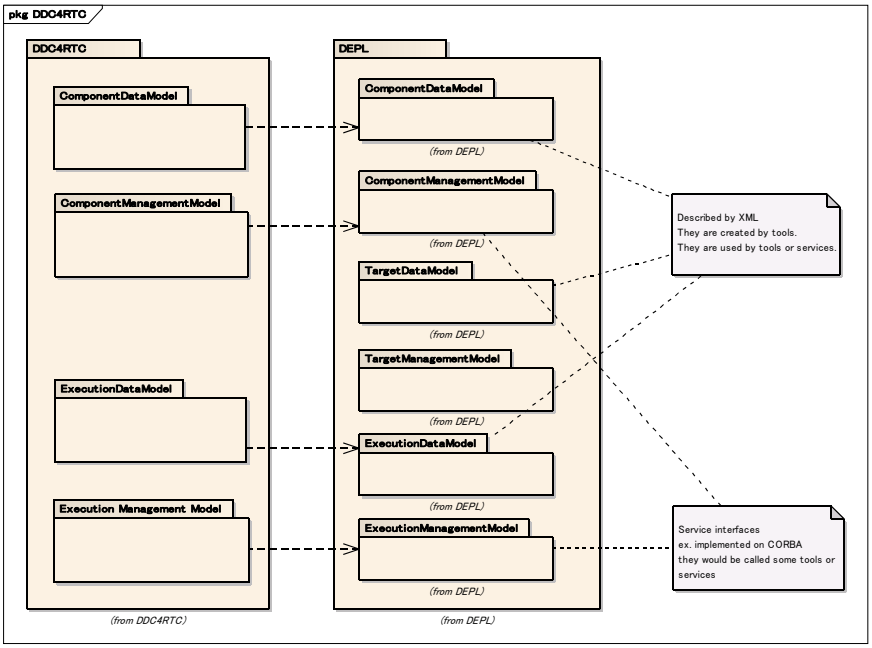

図 22 パッケージダイアグラム：DDC4RTC 標準と DEPL 標準の関係

統合仕様案は DEPL 標準を参照しつつ、拡張・追加した部分についてのみ記述されている。RTS Profile や RTC Profile 同様に MDA に基づき、PIM（プラットフォーム非依存モデル）と PSM（プラットフォーム依存モデル）に分けて定義されている。以下に統合仕様案の目次を示す。

コンポーネントデータモデル

RT コンポーネントの仕様を記述するデータモデルを、コンポーネントデータモデル（Component Data Model）と呼び DEPL のコンポーネントデータモデルを拡張する形で定義している。これは上述の RTC Profile を元に定義されており、RTC 特有のコールバックの実装に関する属性や、バージョン、作成・更新日時、コンポーネントのタイプ等を記述できるようになっている。図 23 にコンポーネントデータモデルのクラス図を示す。

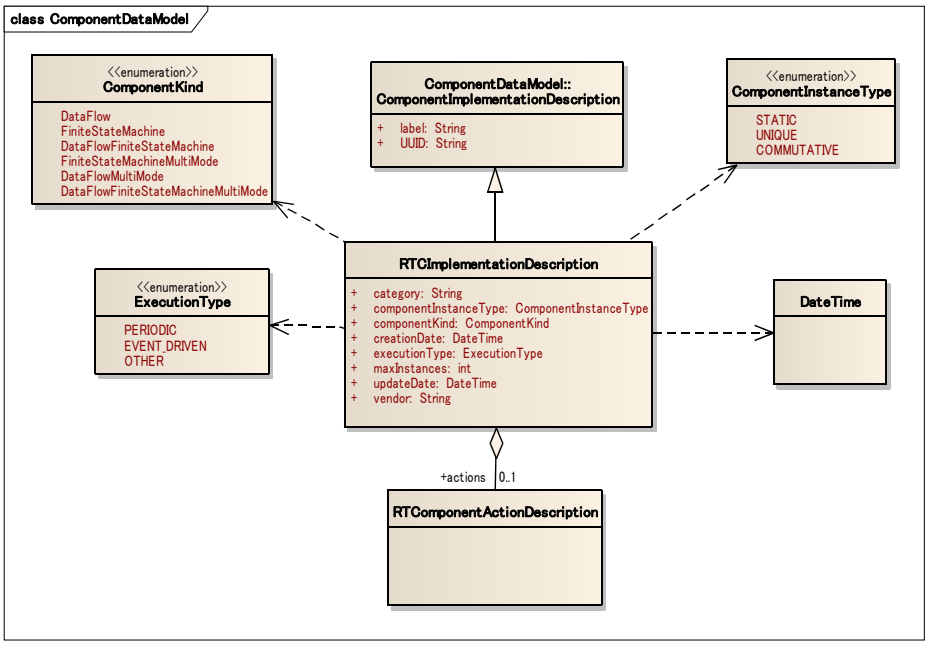

図 23 コンポーネントデータモデル（Component Data Model）

DEPL におけるポートと、RTC におけるポートは若干モデルが異なる。図 24 に示すように、DEPL では、サービスインターフェースそのものをポートと呼び、インタフェース自体がコンポーネントに直接的に属している。一方 RTC においては、すべてのサービスインターフェースはポートと呼ばれる接続を管理する端点となるオブジェクトに属しており、複数のインタフェースをまとめて一つの相互作用端点と見ることができるようになっている。

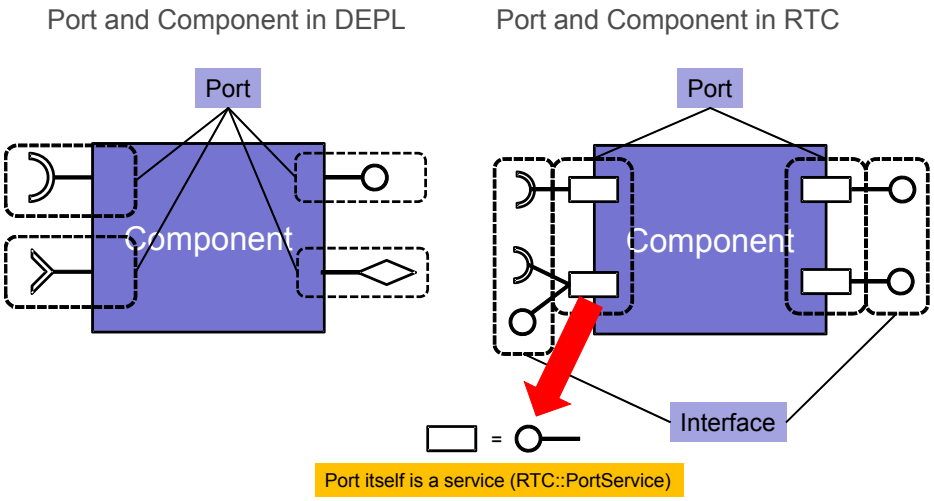

図 24 DEPL と RTC のポートの違い

このようなコンポーネントモデルの若干の差を吸収するために、図 25 に示すように、既存のポート記述に対して拡張を施している。

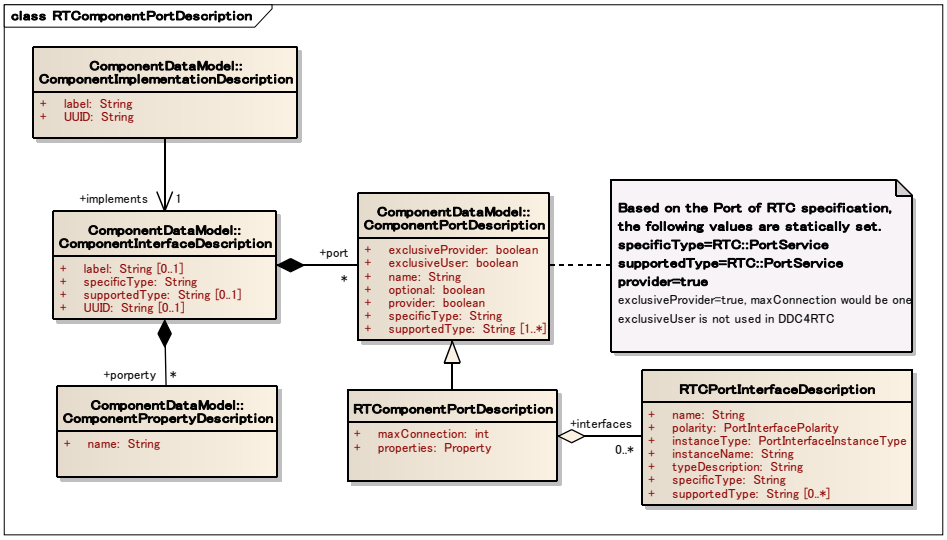

図 25 RTComponentPortDescription クラス図

以上により、DEPL を用いて RTC モデルに基づくコンポーネントを扱うことが可能となった。

ApplicationSupervisor

ApplicationSupervisor はシステムが受け取った何らかのイベントに応じて、事前に定義された状態遷移を行い、システム構成の動的変更を行うためのサービスの一種である。図 26 に ApplicationSupervisor のクラス図を示す。コンポーネントの管理を行う DEPL の ApplicationManager とシステムな以外からのイベント通知を受信するための NotificationService の StructuredPushConsumer を継承して実現されている。

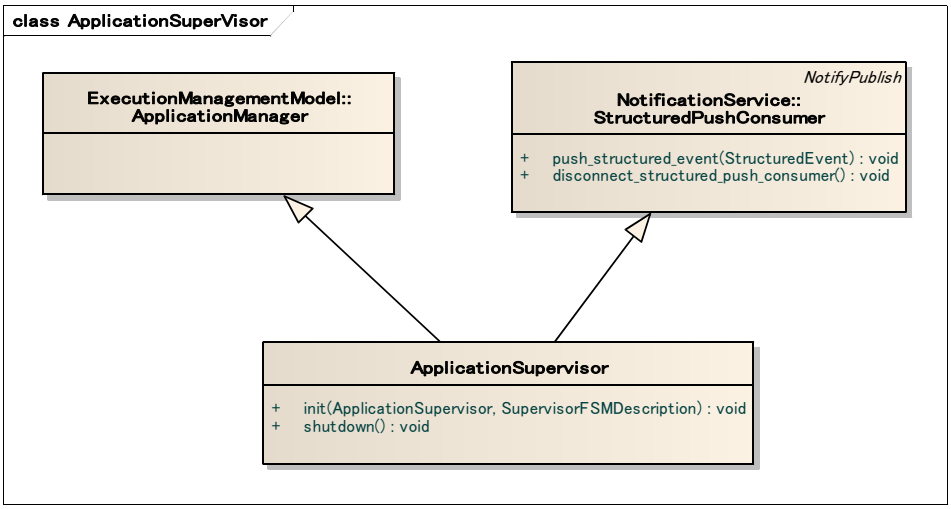

図 26 ApplicationSuperVisor クラス図

図 27 に ApplicationSupervisor のシーケンス図を示す。何らかのイベントが発生すると、内部で管理する状態を遷移させ、遷移に割り当てられたシステム構成を起動するために、コンポーネントの起動、ポート間の接続の切断と再接続、コンフィギュレーション・パラメータの設定等を自動で行う。

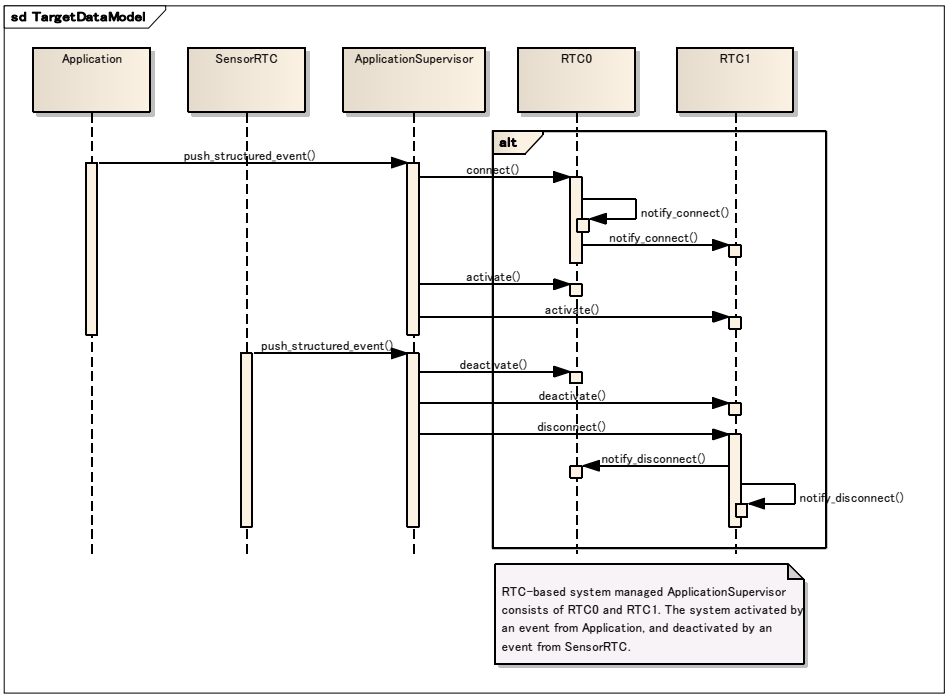

図 27 ApplicationSupervisor のシーケンス図

プラットフォーム依存モデル

DDC4RTC においては、XML および CORBA IDL の PSM を仕様として提供する予定である。

まとめ

策定した RTC 仕様記述方式はプロジェクト参加組織に対して公開すると共に、Web ページ (http://www.openrtm.org)において一般にも公開し、意見・要望などを調査した。本仕様記述方式は、RTCBuilder や RTSystemEditor などで使用され、ツール間のデータ連携に寄与している。RTC Profile の国際標準化活動に関して詳細は、上述の標準化活動の項目で述べたように、RTS Profile および後述の RTC Profile を含め、RT コンポーネントの動的デプロイメントとコンフィギュレーションに関する各種データモデルやサービスインターフェースを標準化する取り組みを平成 20 年 12 月から OMG の Robotics DTF (Domain Task Force)・Infrastructure WG (Working Group)において行ない、DDC4RTC（RT コンポーネントに関する動的配置・設定に関する標準）の標準仕様策定作業における一次提案仕様に本仕様記述方式を取り入れた。

したがって、安定版の知能モジュール仕様記述方式を策定することができ、仕様を本プロジェクト外部に対しても公開することができた。また、国際標準化組織である OMG で DD4RTC の一部として国際標準化がすすめられている。

参考文献

[1] Object Management Group, “Robotic Technology Component Specification Version 1.0”, formal/2008-04-04, http://www.omg.org/spec/RTC/

[2] 産業技術総合研究所 知能システム研究部門,「RTC 仕様記述方式」version 0.2, http://www.openrtm.org/openrtm/ja/node/943

[3] Object Management Group, “Deployment and Configuration of Component-Based Distributed Applications”, formal/06-04-02, http://www.omg.org/spec/DEPL

(a-3) ハードウェア仕様記述方式

ハードウェア仕様記述方式は、RT コンポーネントに関連付けられたデバイスに関する種々の情報であり、RT コンポーネントのメタ情報の拡張情報に関する記述方式である。この情報は、RT コンポーネント作成時、シミュレーション時、システム構成時などに利用される。ハードウェア情報として、

- デバイスメタ情報

- データ型、意味、単位、分解能
- 出力レンジ、定格（センサデバイス等）
- 入力レンジ、定格（アクチュエータ等）
- 機構モデル（構造・形状・物理パラメータ）

等を記述することにした。最終目標としては、安定版のハードウェア仕様記述方式を策定するとともに、仕様を本プロジェクト外部に対しても公開することである。

記述方式の検討

内部委員会にてハードウェア仕様記述方式を検討するとともに、全体委員会に対してハードウェア仕様記述方式草案を提示し意見交換を行った。草案では記述方式は2つの記述方式で構成することとした。センサ、アクチュエータ等のハードウェアコンポーネントの仕様に関する記述方式とそれらをツリー状に組み上げることによって定義される一体のロボットハードウェアの記述方式である。

ハードウェアコンポーネント記述方式

この記述方式はモータや各種センサなどロボットハードウェアを構成するハードウェアコンポーネントの仕様を記述するものであり、これをハードウェアコンポーネント（HWC）プロファイルと呼ぶ。HWC プロファイルは全ての HWC に関して共通の属性から構成される基本プロファイル部分と、HWC の種類別に定義される拡張プロファイルからなる。

基本プロファイルは HWC を一意に識別するための識別子を構成するベンダ名等の情報と動力学シミュレーションを行う際に必要となる質量、重心位置などの動力学パラメータおよび形状データから成る。

拡張プロファイルは以下の各種 HWC それぞれに特有のパラメータで構成される。

1 Motor
2 Gear
3 Accelerometer
4 Gyrometer
5 ForceSensor
6 Camera
7 RangeFinder
8 RangeFinder2D
9 Power

例えば Motor に関してはトルク定数、ロータ抵抗、ロータ慣性等のパラメータが、Camera に関しては画角や画素数等のパラメータが定義される。

以上のハードウェアコンポーネントプロファイルは XML 形式でファイルに格納さ

れる。

ロボットハードウェア記述方式

この記述方式はロボットのリンク機構の構造や各リンクの物理パラメータ等をツリー構造として記述していくものである。

記述方式のベースとしては、科学技術連携施策群の効果的・効率的な推進「次世代ロボット共通プラットフォーム技術の確立」において平成 19 年度末に開発を完了した「分散コンポーネント型ロボットシミュレータ」OpenHRP3 のモデルファイルを採用することとした。このベースに対して、

A) ファイルのエンコーディングをどのように行うか

B) ハードウェアコンポーネントの仕様記述をどのように組み込むか

の検討を行った。

エンコーディングに関しては、

A) VRML97 + プロトタイプ定義による拡張

B) X3D(XML に対応した VRML97 後継規格) + プロトタイプ定義による拡張

C) 独自タグ定義の XML 記述

を候補として検討を行った。当初、比較的新しい標準規格である X3D の採用を検討したが、これには以下のような問題があったため、採用を見送った。

A) 仕様が肥大化しており、対応にコストがかかる。

B) X3D では、仕様の肥大化に付随する問題を避ける意図で、あらかじめいくつかの Subset 仕様がその大きさに関して段階的に定義されており、対応ツールは必要に応じて対応 Subset を選べばよいとしている。しかし、モデルファイルで必要となるプロトタイプ定義が使える Subset は定義されている中でも大きいものになってしまっており、プロトタイプを使おうとすると結局肥大化した仕様を扱う必要が生じる。また、X3D を出力するツールが必ずしも小さいほうの Subset を出力するとも限らない。

C) XML 記述においてプロトタイプ定義したノードを使う際には、記述が非常に煩雑になってしまう。これは X3D として XML のタグを定義した以上、X3D のファイルである ためには、ユーザは独自のタグを使うことはできないからである。

D) 2004 年に ISO の国際標準規格として承認されているが、その後のツール類の対応状 況は VRML97 よりも悪い。これに関しては、A)の問題も関連していると思われる。

X3D のこの状況と比較すると、VRML97 は比較的扱いやすく、従来の OpenHRP シミュレータでの使用実績もあるため、これを採用することとした。ただし、XML 記述されたハードウェアコンポーネント情報を組み込むための拡張が必要になる。

これに関しては、ハードウェアコンポーネント情報を記述したファイルへのリンクを張るための Hardware-Component ノードを定義することにより対応した。
仕様記述方式の統一性を考えると、他の仕様記述が採用する XML による記述とするメリットも大きい。ただし XML に対応した X3D については上で述べた問題があるため、独自にタグを定義した XML によってモデルファイルのエンコーディングを行うことが考えられる。

Modelica の検討

前節で述べた草案を作成した段階で、同様のメカトロシステムの記述する言語として Modelica という言語が存在することが判明した。Modelica とは機械、電気、油圧、熱などのマルチドメインに対応したモデリング言語であり、非営利国際組織である Modelica 協会によって言語仕様の策定やメンテナンスが行われており、その標準ライブラリはフリーで公開されている。Modelica を採用しているシステムとしては CATIA Systems, Dymola, LMS AMESim, MapleSim 等が存在する。
既存の仕様が十分な記述能力を持っているのであれば、その仕様に基づいて作成されているツール群の利用が可能となるため、Modelica の調査を行った。その結果十分な記述能力を持つことがわかったが、一方で従来の資産が継承できなくなる問題が発生することが予想されたため採用を見送った。

IDL によるハードウェア仕様記述

以上のように XML や Modelica などの特定のファイルフォーマットを用いたハードウェア仕様の記述方式を検討してきたが、特定のファイルフォーマットに記述方法を限定してしまうと、従来の資産が使用出来なくなるという問題が発生するため、ハードウェアの仕様記述自体はより抽象度の高いレベルで定義し、実際のファイルフォーマットへのマッピングは実装によって実現することとした。
仕様記述の記述言語として IDL(Interface Definition Language)を用い、記述方式を以下のように定義した。

```
module OpenHRP {
/// ShapeInfo へのインデックス。ShapeInfo に対しては transformMatrix
///に格納された座標変換を適用する。
  struct TransformedShapeIndex {
   /// 変換行列。4x4 同次変換行列の上３行分を Row Major Array で格納したもの。
    DblArray12          transformMatrix;
    /**
       本構造体が LinkInfo に格納されている場合、LinkInfo の
```

```
        inlinedShapeTransformMatrices へのインデックスを格納する。形状がモデルファイルにおいて inline ノードとして別ファイルに記述されている場合、inline ノードを読み込む側における変換のみを集積した行列とする。inline ノードが使われていない場合は -1 とする。
      */
      short inlinedShapeTransformMatrixIndex;
      short       shapeIndex;
  ///< BodyInfo::shapes における ShapeInfo のインデックス。
    };
    typedef sequence<TransformedShapeIndex> TransformedShapeIndexSequence;
    typedef                                 sequence<TransformedShapeIndexSequence>
AllLinkShapeIndexSequence;

    /// センサ情報を格納する構造体。
    struct SensorInfo{
        /*
          センサの種類を表す文字列。現在のところ、以下が定義されている。
          "Force"            - 6 軸力センサ
          "RateGyro"         - レートジャイロセンサ
          "Acceleration" - 加速度センサ
          "Vision"           - ビジョンセンサ
          "Range"                    - 距離センサ
      */
      string            type;
      string            name;           ///< 本センサの識別名
      long              id;             ///< センサの種類ごとのセンサ ID
      DblArray3         translation; ///< センサ設置位置(リンク座標系相対)
      DblArray4         rotation;       ///< センサ設置姿勢(リンク座標系相対)
      FloatSequence   specValues;   ///< 各種仕様値
      string            specFile;       ///< 仕様記述ファイル名

      /// 本リンクに対応する形状情報の変換行列付きインデックス列
      TransformedShapeIndexSequence shapeIndices;
      DblArray12Sequence inlinedShapeTransformMatrices;
    };
    typedef sequence<SensorInfo> SensorInfoSequence;
```

```
/// ハードウェアコンポーネント情報を格納する構造体
struct HwcInfo{
  string          name;          ///< 本 HWC の識別名
  long            id;            ///< HWC の種類ごとの ID
  DblArray3       translation; ///< HWC 設置位置(リンクローカル座標)
  DblArray4       rotation;    ///< HWC 設置姿勢(リンクローカル座標)
  string          url;           ///< HWC プロファイルの URL

  /// 本 HWC に対応する形状情報の変換行列付きインデックス列
  TransformedShapeIndexSequence shapeIndices;
  DblArray12Sequence inlinedShapeTransformMatrices;
};
typedef sequence<HwcInfo> HwcInfoSequence;

/// セグメントの情報を格納する構造体。複数個のセグメントノードを持つリンク
/// を GUI から編集するために使用
struct SegmentInfo{
  string        name;              ///< セグメント名
  double        mass;              ///< 質量
  DblArray3     centerOfMass;      ///< 重心位置
  DblArray9     inertia;           ///< 慣性行列
  DblArray12          transformMatrix;
  sequence<short> shapeIndices; ///   < TransformedShapeIndex のインデックス列
};
typedef sequence<SegmentInfo> SegmentInfoSequence;

/// 各リンクの情報を格納する構造体。
struct LinkInfo{
  string          name;          ///< リンク名
  short           jointId;       ///< 関節識別値
  string          jointType;     ///< 関節タイプ
  double          jointValue;    ///< 関節初期値
  DblArray3       jointAxis;     ///< 関節軸(リンクローカル座標)
  DblSequence     ulimit;        ///< 最大関節値
  DblSequence     llimit;        ///< 最小関節値
```

```
    DblSequence     uvlimit;          ///< 最大関節速度値
    DblSequence     lvlimit;          ///< 最小関節速度値
    DblArray3       translation;    ///< ローカル座標系原点(親リンク相対)
    /// ローカル座標系姿勢(親リンク相対)。回転軸(x, y, z) + 回転角度の並びの
    ///サイズ４の配列
    DblArray4       rotation;
    double          mass;             ///< 質量
    DblArray3       centerOfMass;  ///< 重心位置
    DblArray9       inertia;          ///< 慣性行列
    double          rotorInertia;    ///< ロータ慣性
    double          rotorResistor; ///< ロータ抵抗
    double          gearRatio;        ///< ギア比
    double          torqueConst;     ///< トルク定数
    double          encoderPulse;    ///< エンコーダパルス
    short           parentIndex;      ///< 親リンクインデックス
    ShortSequence childIndices; ///< 子リンクインデックス列

    /// 本リンクに対応する形状情報の変換行列付きインデックス列
    TransformedShapeIndexSequence shapeIndices;
    short AABBmaxDepth; ///< 形状データの AABBtree の階層の深さ＋１
    short AABBmaxNum;     ///< 形状データの AABBtree の BoundingBox の最大個数
    /// shapeIndices の inlinedShapeTransformMatrixIndex によって指し示される
    /// 行列リスト
    DblArray12Sequence inlinedShapeTransformMatrices;
    SensorInfoSequence sensors; ///< 本リンクに設置されたセンサの情報
    HwcInfoSequence hwcs;
    SegmentInfoSequence segments;
    /// アクチュエータ・ギア等の仕様記述ファイル名リスト
    StringSequence specFiles;
};
typedef sequence<LinkInfo> LinkInfoSequence;
/// 物体形状情報を格納する構造体。
enum ShapePrimitiveType { SP_MESH, SP_BOX, SP_CYLINDER,
                          SP_CONE,SP_SPHERE, SP_PLANE };
struct ShapeInfo{
    /**
```

本 Shape が VRML の inline ノード内に格納されている場合は、inline されている VRML ファイルへのパスを格納する。inline ではなく直接メインの VRML ファイル内に形状が記述されていた場合は、本フィールドは空とする。
*/
string url;
/**
オリジナルの VRML モデルファイルにおけるプリミティブの種類を表す。クライアントは描画においてこの情報を利用することができる。ただし、primitiveType が MESH 以外のときも、プリミティブをメッシュに展開した際の幾何データ(vertices など)は持っているものとする。
*/

ShapePrimitiveType primitiveType;
/**
primitiveType が MESH 以外のとき、プリミティブの形状に関わるパラメータを格納する。各プリミティブにおける配列要素とパラメータの対応は以下とする。
- BOX 0-2: x, y, z のサイズ
- CYLINDER 0: radius, 1: height, 2: top, 3: bottom, 4: side
bottom, side, top については値が 0 のとき false、それ以外は true とする。(CONE に関しても同様。)
- CONE 0: bottomRadius, 1: height, 2: bottom, 3: side
- SPHERE 0: radius
*/
FloatSequence primitiveParameters;
/// 表面メッシュを構成する頂点データ。連続する３要素が頂点位置の３次元
/// ベクトルに対応する。
FloatSequence vertices;
/**
表面メッシュを構成する三角形における頂点の組み合わせを格納したデータ。各要素は vertices に格納された頂点のインデックスを表し、連続する３要素によってメッシュを構成する三角形を指定する。メッシュ構成面は必ず三角形で表現されるものとする。なお、メッシュの表裏を区別する必要がある場合は、連続する３要素が反時計回りとなる面を表とする。
*/
LongSequence triangles;

```
  /// 本 Face に対応する AppearanceInfo の BodyInfo::appearances における
  /// インデックス。
 long appearanceIndex;
};
typedef sequence<ShapeInfo>        ShapeInfoSequence;
/// 表面の見え情報を格納する構造体。
struct AppearanceInfo{
  /// 本 Appearance に対応する MaterialInfo がある場合、BodyInfo::materials
  /// におけるインデックス。なければ -1。
  long                materialIndex;
  /**
      法線データ。連続する３要素を x, y, z とし、一法線ベクトルに対応。
      この配列のサイズが 0 の場合、法線はクライアントが必要に応じて生成し
      なければならない。
    */
  FloatSequence normals;
  /**
      法線対応付けデータ。normalPerVertex が true なら、ShapeInfo の
    vertices の並びと対応させる。normalPerVertex が false なら、ShapeInfo
    における三角形ポリゴンの並びと対応させる。normals があって
    normalIndices のサイズが 0 の場合、normals の要素を頂点または面に 1 対 1
    対応させる。
  */
  LongSequence    normalIndices;
  boolean           normalPerVertex;
  boolean           solid;
  float               creaseAngle;
  /**
    色データ。連続する３要素を R,G,B とし一色に対応。各要素の値の範囲は 0 か
    ら 1.0。この配列のサイズが 0 の場合、色は materialInfo のものになる。
  */
  FloatSequence colors;
  /**
      色対応付けデータ。colorPerVertex が true なら、ShapeInfo の vertices の
      並びと対応させる。colorPerVertex が false なら、ShapeInfo における三角
      形ポリゴンの並びと対応させる。colors があって colorIndices のサイズが 0
```

```
      の場合、colors の要素を頂点または面に 1 対 1 対応させる。
    */
    LongSequence    colorIndices;
    boolean         colorPerVertex;
    /// テクスチャデータ。BodyInfo::textures におけるインデックス。
    /// 対応するテクスチャがなければ、-1。
    long            textureIndex;
    FloatSequence textureCoordinate;
    LongSequence    textureCoordIndices;
    DblArray9    textransformMatrix;
};
typedef sequence<AppearanceInfo> AppearanceInfoSequence;
/**
    表面材質情報を格納する構造体。各要素は VRML の Material ノードと同様。
    全ての変数の値の範囲は 0.0 ～ 1.0。
*/
struct MaterialInfo{
    float           ambientIntensity;
    FloatArray3     diffuseColor;
    FloatArray3     emissiveColor;
    float           shininess;
    FloatArray3     specularColor;
    float           transparency;
};
typedef sequence<MaterialInfo> MaterialInfoSequence;
/// テクスチャ情報を格納する構造体。各要素は VRML の PixelTexture ノードと同様。
struct TextureInfo{
    /**
      テクスチャの画像イメージ。VRML の SFImage から、先頭の width, height,
      num components を除いたものと同様。width, height, num components に対
      応する値は本構造体の width, height, numComponents で指定。元のデータ
      が url 指定の場合は、url フィールドに画像ファイルの位置が格納される。こ
      の場合、モデルローダ側で画像の展開が行われなかった場合は、image フィ
      ールドのサイズは 0 となっており、クライアントはファイル名からテクスチ
      ャを獲得する必要がある。image フィールドのサイズが 0 でなくて、url
      のサイズも 0 でない場合は、クライアントは好きな方のやり方でテクスチ
```

```
        ャ画像を獲得すればよい。
    */
    OctetSequence image;
    short           numComponents;
    short           width;
    short           height;
    boolean         repeatS;
    boolean         repeatT;
    string            url;
};
typedef sequence<TextureInfo> TextureInfoSequence;
/// 形状データー式を格納するオブジェクト。
interface ShapeSetInfo{
    /**
        表面の形状と見えの情報を格納する ShapeInfo のシーケンス。LinkInfo において、Link に対応する情報が本シーケンスのインデックスとして指定される。
    */
    readonly attribute ShapeInfoSequence shapes;
    /// Appearance 情報のシーケンス。ShapeInfo において、本シーケンスのッ
    /// インデクスが指定される。
    readonly attribute AppearanceInfoSequence appearances;
    /// Material 情報のシーケンス。AppearanceInfo において、本シーケンスの
    /// インデックスが指定される。
    readonly attribute MaterialInfoSequence materials;
    /// Texture 情報のシーケンス。AppearanceInfo において、本シーケンスの
    /// インデックスが指定される。
    readonly attribute TextureInfoSequence textures;
};
/// 物体モデル情報へのアクセスを提供するインタフェース。
interface BodyInfo : ShapeSetInfo{
    readonly attribute string name; ///< モデル名
    readonly attribute string url; ///< モデルファイルの URL
    /// Humanoid ノードにおける info フィールドに記述されたテキスト。
    readonly attribute StringSequence info;
    /**
```

```
        リンクの機構情報を全リンクについて格納したデータ。本シーケンスにおける LinkInfo の並びは、linkIndex(モデルファイルにおける JointNode 出現順。jointId とは異なる。)の順。
      */
      readonly attribute LinkInfoSequence links;
      /**
        リンクに対応する ShapeInfo のインデックス配列を、全てのリンクに関して格納した配列。リンクの並びは linkIndex の順とする。
      */
      readonly attribute AllLinkShapeIndexSequence linkShapeIndices;
    };
    interface SceneInfo : ShapeSetInfo{
      readonly attribute string url; ///< 形状ファイルの URL
      /// LinkInfo の shapeIndices と同じ
      readonly attribute TransformedShapeIndexSequence shapeIndices;
    };
};
```

まとめ

本記述方式は、動力学シミュレータと共に一般に公開・配布されており、本プロジェクトで開発した動力学シミュレータ、ハードウェア設計支援ツール、移動動作設計ツールは本記述方式に基づいて実装されている。これらのツールは具体的なファイルフォーマットとしては VRML97 を用いている。また、これまでに東京大学、ゼネラルロボティックス（株）により COLLADA へのマッピングも実装が行われている。

(a-4) シナリオの記述方式

作業シナリオとは、時間及びイベントによりシステム構成を動的に変更しつつ、ロボットに一連の動作を行わせ目的作業を達成するための、ロボットの動きや手順を記述したファイルである。具体的には動作設計ツールにより生成される「動作記述」と、シナリオ設計ツールにより生成される「動作制御記述」に分けられる。

(A) 動作記述

「動作記述」はデータを主体とした記述方式であり、作業シナリオ中から呼び出

されるロボットのある定められた動作（動作パターン）の記述方式を定めるものである。基本的には、対象とするロボットの各関節の角度軌道を全関節分記述する。動作記述方式の策定に当たっては、本プロジェクトの各研究項目実施機関で開発、使用されるモジュール群での利用を勘案し、またそれらの機関での試用からのフィードバックを活用する。最終目標は、安定版の動作記述方式を策定するとともに、仕様を本プロジェクト外部に対しても公開することである。

概要

「動作記述」はデータを主体とした記述方式であり、作業シナリオ中から呼び出されるロボットのある定められた動作（動作パターン）の記述方式を定めるものである。

本仕様の策定は開発項目のひとつである「動作パターン設計ツール」の開発を通して行った。ツールにおける動作パターンデータの保存・読み込み機能の実装を通して開発することで、実際の利用において扱いやすく効率的な記述方式を定めるというのがそのねらいである。

実際に策定した記述方式の概要は以下の通りである。

- 基本フォーマットとして YAML を採用
- YAML において構造化されたノードとして、ロボットの関節角軌道やリンクの位置姿勢の軌道、およびそれらの属性を記述する。
- 各データのタイプは特定の文字列によって認識され、新たなタイプのデータを容易に追加し運用できる設計としている。

基本フォーマット

本項目の目的を達成するためには、CSV 形式のような単なるデータの羅列ではなく、何らかのかたちで構造化された記述方式であることが必要となる。これについて、新たな構造化フォーマットを考案することも考えられなくはないが、それよりは既に広く使われている構造化フォーマットを採用し、その上に動作記述のためのフォーマットを構築する方が、記述の理解や運用を容易にするにあたって有効である。

そのような基本とする構造化フォーマットの候補としては、当初 XML が最有力であった。何故なら、XML は Web を中心として既に広く使われているよく知られたフォーマットであり、読み書きのためのライブラリも充実しており、その構造化記述能力も本項目の目的を達成するにあたって十分なものであったからである。

しかしながら、実際に動作パターン設計ツールにおいて XML 形式での動作保存・読み出し機能の実装を進めたところ、XML 形式を採用した場合以下のような

問題があることが判明した。

- DOM, SAX 等の API に基づく標準的な XML 読み込みライブラリを用いた読み込み処理のプログラミングは実際には大変煩雑で多くのコーディングを要する。
- 上の問題とも関連して、ファイルの読み込みは比較的重い処理となってしまう。
- 実際の XML のテキストファイルは人間が読みやすく編集しやすいものではない。

このような問題が生じる大きな理由は、XML がもともとマークアップ言語として設計されたものであることによる。マークアップ言語は基本的に、何らかのデータ（テキスト等）があった上で、その各所にマークアップして属性を付加するという発想で設計されている。これは各種文書を格納するには適切な形式であるものの、比較的シンプルなデータ要素を構造化していく用途に対しては、必ずしも適切なものではない。XML はそのようなデータ構造もカバーできるようになっているが、マークアップ言語としての特性から記述の自由度が高く、そのようなデータ構造に対してはオーバースペックな仕様を持っているとも言える。そしてそのオーバースペックな部分が上記の問題の原因となっている。

読み込み処理のプログラミングが煩雑になる問題は、プラットフォーム側で読み込みのライブラリを提供することである程度は解決可能である。しかし、複数の言語に対してそのようなライブラリを開発し保守するのはそれなりのコストがかかる。また、プラットフォームが対応していない言語にて動作記述を扱いたい場合には、結局ユーザがあらたに読み込み処理を記述する必要がある。このことを考慮すると、フォーマット自体が目的に対して十分シンプルでプログラミングにおいても扱いやすいことは大変重要である。このことと他の２つの問題を考慮すると、必ずしも XML が適切とは言えない。

もちろん、XML の普及度や知名度は標準プラットフォームとして重要な要素になり得る。しかし XML の上に構築するデータ形式によって結局個別の処理コードが必要になるのであるから、これらの要素は本プラットフォームの目的に対してさほど実質的な意味をもつものではなく、イメージ的なものであると言える。

YAML 形式の採用

XML に関する以上の考察から他の適切なフォーマットを検討したところ、YAML 形式が有力な候補となった。YAML はマークアップ形式ではなく、データ記述の構造的に焦点を当てた XML よりシンプルなテキストフォーマットである。基本的には、数値、文字列等のいくつかのスカラー型を、リストとマップの組み合わせで構

造化できる仕様となっている。記述自体もインデントもしくはブレースを用いたシンプルなものとなっており、実際のテキストファイルも見やすいものとなっている。また、YAML は JavaScript 言語のデータ構造記述を抽出した”JSON”フォーマットのスーパーセットでもある。JSON も Web を中心として既に広く利用されている汎用的なフォーマットとなっており、YAML においてもその性質が受け継がれており、JSON 部分の記述に絞ることで対応ライブラリなどに関する汎用性をさらに向上させることも可能である。

プログラミングについては、多くの言語が YAML もしくは JSON フォーマットの読み込みをサポートしている。XML とは異なり、読み込んだ内容はそのまま各言語におけるリストやマップに格納された構造化データとなるため、その後のデータ抽出も容易である。具体的には、Java, Python, Ruby などの言語が標準で読み込みをサポートしている。また C/C++についてはパースライブラリがいくつか存在し、さらにそれを用いて構造化データとして読み込む C++ライブラリを動作パターン設計ツールのライブラリとして提供している。

読み込み処理の効率性についても XML より効率的なものとなっており、また実際のファイルの可読性についても一般的に XML より高いものとなっている。

以上の考察により、動作記述の基本フォーマットとして YAML を採用するに至った。

仕様

実際の仕様は以下のとおりである。なお、YAML 自体の仕様については、http://yaml.org にて配布されている仕様書を参照のこと。

トップレベル要素：

以下の要素を持つマップ型とする。

type: BodyMotion

本データファイルが動作記述であることを示すマップ要素。

components: 動作データの各コンポーネントを格納するシーケンス。

“components”シーケンスの要素:

以下のキーを有するマップ型とする。

type: コンポーネントの型を表す文字列を格納する。

content: コンポーネントの内容を表す文字列を格納する。

frameRate: フレームレートの数値を格納する。

numFrames: フレーム数を格納する。

numParts: コンポーネントのパート数を格納する。

frames: 動作データをフレーム順で並べたシーケンス型で格納する。

各フレームはさらに各パートを格納するシーケンス型とする。

“components”の “type” に対して定義されたデータタイプ：

MultiValueSeq: １フレームあたり浮動小数値が複数（パート数分）格納された
データ型。関節角軌道などが対応する。

MultiValueSeq に対して通常使われる “content”タイプ：

JointPosition: 関節角軌道

MultiSE3Seq: １フレームあたり位置と姿勢の組が複数（パート数分）格納された
データ型。リンクの位置姿勢軌道などが対応する。

MultiSE3Seq に対して通常使われる “content”タイプ：

LinkPosition: リンクの位置姿勢の軌道。numParts がリンク数に対応。

MultiSE3Seq に対して追加される属性キー。

format: 位置姿勢の記述方式を表す文字列。現在以下が定義されている。

“XYZQXQYQZQW”: 位置姿勢を、位置 X 座標、Y 座標、Z 座標、
姿勢クオタニオン X 要素、Y 要素、Z 要素、W 要素の順に並べる

“XYZRPY”: 位置姿勢を、位置 X 座標、Y 座標、Z 座標、
オイラー角ロール要素、ピッチ要素、ヨー要素の順に並べる

MultiSE3Seq における”frames”の記述：

「「「位置姿勢の各要素を format に従って並べたシーケンス」をパート数分
並べたシーケンス」をフレーム数分並べたシーケンス」とする。

拡張的な“components”要素の例:

type: ZMP ： ゼロモーメントポイントの軌道を格納する。

その他、ユーザが必要に応じて”components”定義を拡張し、任意のデータを格納することが可能である。

実際のファイルの例

関節数 9 の PA10 型マニピュレータの関節角軌道とベースリンクの位置姿勢軌道を記述した例の一部を以下に示す。（PA10 はベース固定型のマニピュレータのため、リンクの位置姿勢については全て原点における基準姿勢となっている。）

```
type: BodyMotion
components:
  -
    type: "MultiValueSeq"
    content: "JointPosition"
    frameRate: 1000
    numFrames: 9802
```

numParts: 9

frames:

  - [0, 0.8, 0, 0.8, 0, 0.8, 1.57, -0.02, 0.02]

  - [-1.97358397e-10, 0.800005396, -1.40737758e-09, 0.799999087, 5.62283517e-09, 0.799996667, 1.57, -0.0200000032, 0.0199999968]

  - [-6.57865694e-10, 0.800017988, -4.69118959e-09, 0.799996956, 1.87427829e-08, 0.799988888, 1.56999999, -0.0200000105, 0.0199999895]

  - [-1.69728014e-09, 0.80003754, -8.79723376e-09, 0.799993824, 3.7761151e-08, 0.799976848, 1.56999997, -0.020000021, 0.019999979]

  - [-3.86731477e-09, 0.800063641, -1.1883735e-08, 0.799990069, 5.98847256e-08, 0.799960862, 1.56999996, -0.0200000326, 0.0199999674]

  - [-7.65114255e-09, 0.800095844, -1.22456006e-08, 0.799986097, 8.24597226e-08, 0.799941292, 1.56999994, -0.0200000434, 0.0199999566]

  - [-1.13293439e-08, 0.800133322, -1.30944252e-08, 0.799982963, 1.08604563e-07, 0.799918109, 1.56999993, -0.0200000521, 0.0199999479]

  - [-1.36152256e-08, 0.800175421, -1.68277875e-08, 0.799981418, 1.40620736e-07, 0.799891464, 1.56999991, -0.0200000581, 0.0199999419]

  - [-1.40161691e-08, 0.800221646, -2.41812465e-08, 0.799981935, 1.7893744e-07, 0.799861683, 1.56999989, -0.0200000617, 0.0199999383]

  - [-1.24977426e-08, 0.800271576, -3.49259944e-08, 0.799984853, 2.22885144e-07, 0.799829159, 1.56999987, -0.0200000636, 0.0199999364]

  - [-9.23578975e-09, 0.800324833, -4.8402442e-08, 0.799990442, 2.71328608e-07, 0.799794315, 1.56999985, -0.0200000646, 0.0199999354]

～

-

type: "MultiSE3Seq"

content: "LinkPosition"

frameRate: 1000

numFrames: 9802

numParts: 1

format: XYZQXQYQZQW

frames:

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

  - [[0, 0, 0, 0, 0, 0, 1]]

～

利用状況

本記述形式は動作パターン設計ツールの動作パターンデータ読み込み・保存機能において採用しており、これを用いることで簡単に本記述形式を扱うことが可能である。具体的には、ツール上でキーフレーム機能を使って作成した動作を本記述形式で出力したり、本記述方式で記述された動作データを読み込んで 3D ビュアーやグラフ等でその動作を確認することが可能である。また、動作パターン設計ツールのプラグイン拡張機構を用いて、他の形式の動作データに対する読み込み・保存機能を実装することにより、他形式とのデータの変換も比較的容易に可能となる。実際、この枠組みを用いて、ヒューマノイドロボットの制御システムのひとつである"HRPSYS"で長年使われてきた動作パターン記述形式との相互変換も可能となっている。二足歩行ヒューマノイドロボットにおいては制御システムが目標関節角軌道に加えて ZMP 軌道などの付加的な情報も必要とすることが多いが、そのような ZMP などの付加的な情報も本記述形式を用いて自然に記述することが出来ている。さらに、動作パターン設計ツールのプラグイン拡張機能により、新たなロボット用ツールを開発した場合、基本的にそのツールも本記述形式に対応したものとなる。動作パターン設計ツール上に新たにプラグインとして構築されたツールとしては、把持計画のための"graspPlugin"があり、このツールで計画した動作も本記述形式で出力可能である。

このように、本記述形式は本プラットフォームにおいて標準的な動作記述方式として利用されている。

まとめ

前述の利用状況に述べたように、本動作記述は、動作パターン設計ツールのファイルファーマットとして配布されおり、記述法に関しても公開されている。この記述方式を利用して、把持計画のための知能モジュールも開発されている。

(B) 動作制御記述

「動作制御記述」は時刻やイベントによるロボット制御のための制御構造を記述したファイルであり、シナリオ設計ツールにより生成される。RT コンポーネントや RT コンポーネントで構成されるサブシステムからのイベントの取得や、イベントや時刻によるシステム構成の変更、上記の動作記述の呼び出しによるロボットの制御等の制御構造を記述する。このような動作制御のための作業シナリオの記述方式を策定する。策定に当たっては、本プロジェクトの各研究項目実施機関で開発、使用されるモジュール群での利用を勘案し、またそれらの機関での試用からのフィードバックを活用する。本研究開発の最終目標は、安定版の動作制御記述方式を策定

するとともに、仕様を本プロジェクト外部に対しても公開することである。

動作制御記述の概要

動作制御記述の概要は以下のとおりである。本仕様に基づく RT コンポーネントはその役割によって

- リクエスタ　　要求を出す側
- プロバイダ　　要求を受け取り処理する側

の２種類がある。

それぞれ、少なくともメッセージをやり取りするための入力データポートと出力データポートを１つずつ装備する。

リクエスタとプロバイダは当事者 RT コンポーネント間の役割から見た定義であって、リクエスタが他の RT コンポーネントから見るとプロバイダになる場合もあるし、プロバイダが他の RT コンポーネントから見るとリクエスタになる場合もある。

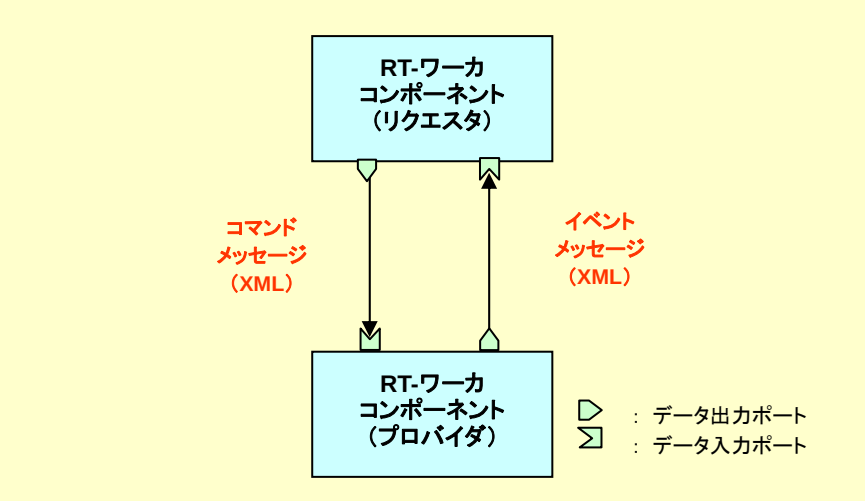

図 28　RT-ワーカコンポーネント間の通信

本仕様に基づく RT コンポーネント間でやり取りするメッセージには大きく次の 2 種類がある。

- コマンド　　リクエスタ→プロバイダに送られる要求
- イベント　　プロバイダ→リクエスタに送られる通知

コマンドは、プロバイダにおける要求入力であり、プロバイダが提供する機能の実行要求を積載する。コマンドのメッセージの中に、機能の名前やその引数が収容される。イベントは、プロバイダにおける結果(事象)出力であり、プロバイダにおける処理の実行結果を積載する。イベントのメッセージの中に、それが伝える事象の名前や付帯する引数が収容される。イベントはその通知先を特定の RT コンポーネ

ントに限定することも出来るし（ユニキャスト）、複数の通知先に同報通知することも出来る（マルチキャスト）、さらに通知先を特に指定しないことも出来る（ブロードキャスト）。
プロバイダは、コマンドの受理した際、あるいは、コマンドによって要求された処理を完了した際、のいずれかの時点で、リクエスタにレスポンスと呼ばれるメッセージを返す必要がある。前者は、リクエスタをプロバイダでの処理の終了に同期させない場合に、後者は同期させたい場合に用いる。
レスポンスは特殊なイベントとして定義される。コマンドに対する結果を戻り値としてリクエスタに通知する。メッセージが積載する名前はコマンド名と同名である。通知先は通常コマンドを発行したリクエスタだけに限定される（複数のコンポーネントに通知する同報の運用は可能）。リクエスタがレスポンスを受け取るかどうかも、リクエスタの自由である。結果を受け取るまで次の処理を待ち合わせる場合には同期型の呼び出しとなり、待ち合わせの必要がない場合は非同期型の呼び出しとなる。非同期型の場合、さらにレスポンスをどこかで受け取るか、棄却するかの選択がある。
メッセージのデータ型は文字列型で XML 形式である。以下に UML 表記で示す。

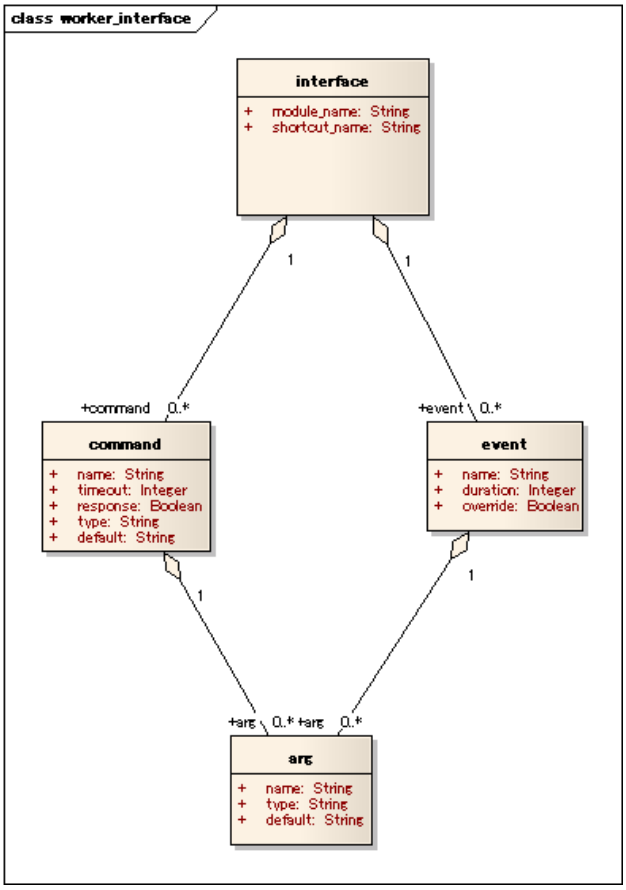

図 29 worker_interface のクラス図

RT コンポーネントがプロバイダとして提供する機能を上記のインタフェース仕様に基づいて定義する。定義はコマンドとイベントに対して行なう。定義書式は XML で供給され、必要により、リクエスタ／プロバイダの双方がこれを読み込み利用する。RT コンポーネント作成者は、このインタフェース仕様に基づいて、RT コンポーネントのインタフェース定義を作成し、コンポーネントと共に利用者に配布する。インタフェース定義は以下の用途に利用される。

- シナリオ開発ツールは、シナリオプログラマが作成したシナリオをパースする際に、このインタフェース定義を読み込み、シナリオと照合することで、シナリオの妥当性検証を行う。
- シナリオ実行系は、シナリオを実行する際に、このインタフェース定義に基づいて、メッセージを構成し、RT ワーカコンポーネントに送信する。

まとめ

上述のような動作制御記述方式を詳細ドキュメント(表題 RT-Worker Component Specification, 英文、30 ページ)にまとめた。この仕様に基づいて、後述の作業シナリオ作成ツールを構築しており、同ツールのリリースパッケージに同ドキュメントを同梱して一般公開を行った。

(a-5) ロボットサービス仕様記述方式

RT コンポーネントに基づく RT システムが実現するサービスを記述するために、ロボットサービス仕様記述方式を策定する。ロボットサービスは UML もしくは SysML で記述し，コンテキスト図、ユースケース図などを用いて、ロボットが提供しようとするサービスとその前提条件を記述する。またサービスを複数の機能に分解して RT コンポーネントとして実装可能な要素に対応させる。

ロボットサービス仕様記述方式の最終的な成果物は、UML もしくは SysML で記述し、RT コンポーネントと対応可能なデータモデルであり、最終目標は、RT コンポーネントの仕様記述に変換可能なロボットサービス仕様記述方式について安定版の仕様を策定し、本プロジェクト外部に対して公開する。また OMG などの標準化組織での国際標準化を目指すことである。

ロボットサービス仕様記述方式の成果概要

知能ロボットを構成する内部的なモジュールはそれぞれ他のモジュールに対してある種のサービスを提供し、それらの集合体は最終的にユーザや環境に対してサービスを提供する。システムをサービスの集合体として体系的に記述し、ロボットシステムの設計・開発時に役立てることができれば、ユーザや外部システムからの要求

を的確に把握することができ、システム開発をより効率的に行うことが可能になる。そこで、RT コンポーネントで構成される RT システムに対して、システムが内包するサービスおよびシステムが提供するサービスを記述方法について検討を行った。ソフトウェアの仕様一般を形式的に記述する方式としては UML（Unified Modeling Language）が知られているが、近年、ハードウェア、ソフトウェア、情報、人、手続、設備を含む複雑なシステムを定義、分析、設計および検証するための汎用的な図式モデリング言語として SysML[1]が注目されている（図 30）。SysML は UML2.0 の一部（サブセット）を利用しつつ、システム記述に必要ないくつかの新しい記述方式を追加したものである（図 31）。

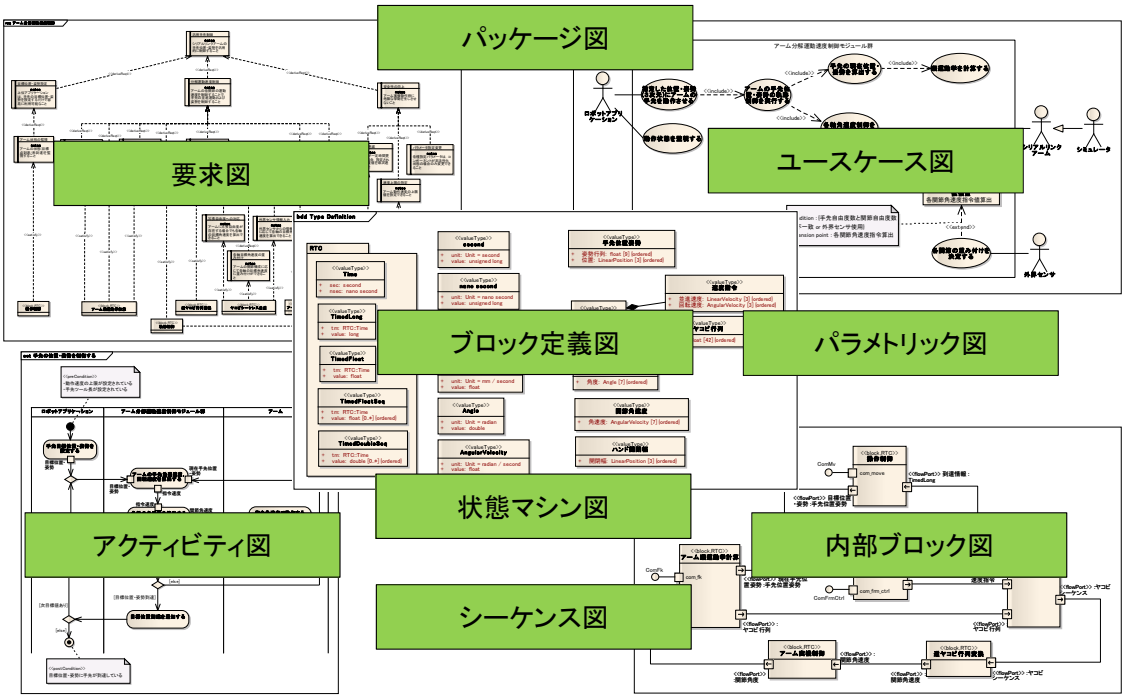

図 30 SysML および 9 種類のダイアグラム

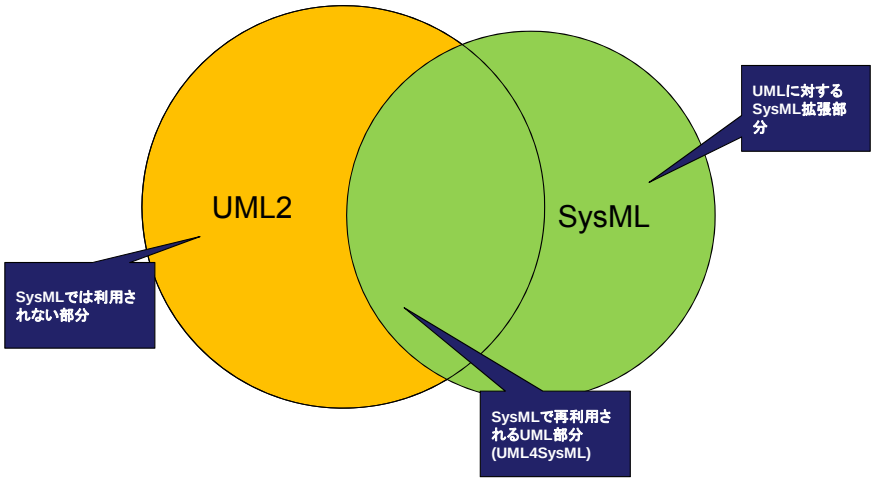

図 31 UML と SysML の関係

ロボットシステムは、ソフトウェアのみならずさまざまなエンジニアリング分野（機械、電気、ソフトウェア）にまたがるシステムであり、特にサービスロボットにおいてはシステムに加えて人（ユーザ）や環境・設備についても考慮し、設計・実装していかなければならない。そこで、SysML を利用し、ロボットが提供するサービス（ロボットサービス）をコンセプトの段階から、ユースケース、要件定義、システムの振る舞い記述、サービスを実現する機能への分割などを SysML により記述し、これをシステム設計・開発プロセスに利用する手法について検討を行った。

SysML モデル化の試み

既存のいくつかのシステムに対して、実際に SysML モデル化を行い、ロボットシステムやサービスがどのようにモデルとして記述可能か、またそれらの記述をどのようにシステム設計・開発プロセスに生かせるかの検討を行った。

ここでは、例の一つとしてアームの分解速度制御システムについて概要を述べる。アームの分解速度制御システムは、入力されたアームの手先速度を、関節角速度に分解し、シリアルリンク型のアームの各関節のモータを制御するシステムである。この例では、シミュレータと実機を切り替えて制御できるシステムを実現することを目指した。まず、このシステムに対する要求を SysML の「要求図」（図 32）として記述する。要求図では、このシステムの目的「シリアルリンクアームの手先位置姿勢を汎用的に制御すること」から出発して、より細かな要件へと分解していく。分解された要件は最終的にいくつかの機能ブロックとして定義される。本研究ではこれらのブロックは RTC として実現されることになる。

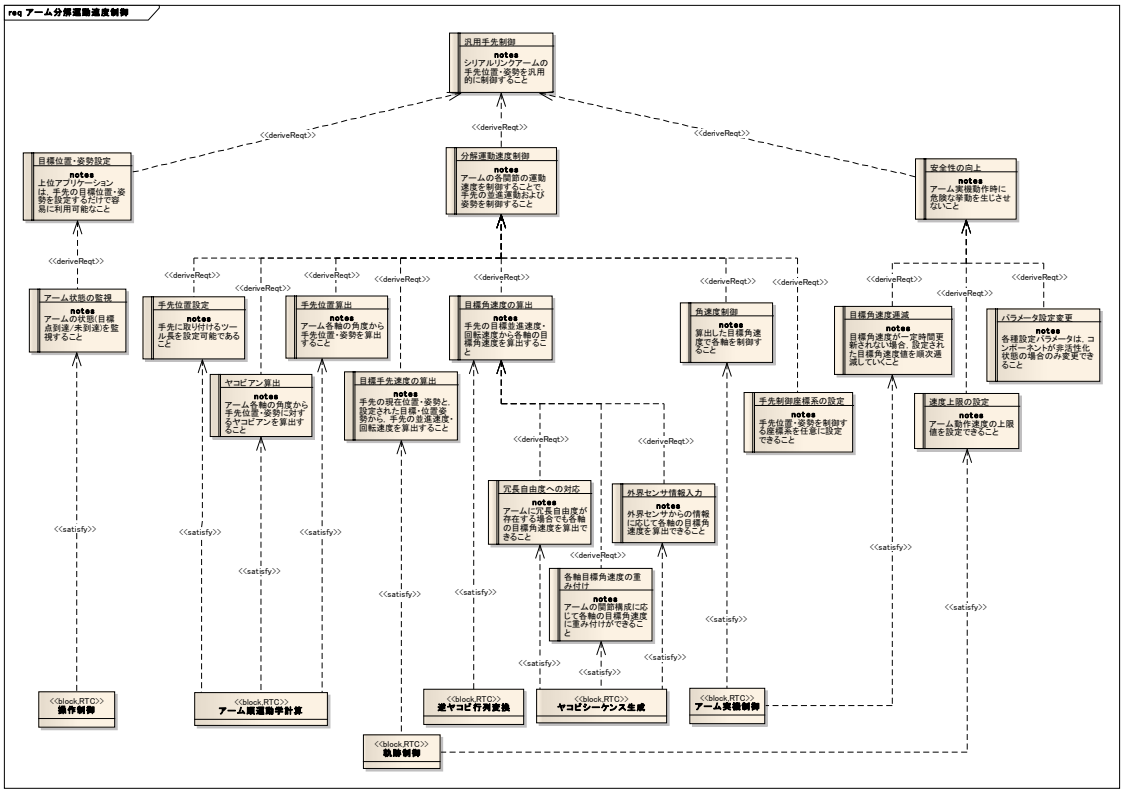

図 32 アームの分解速度制御「要求図」

アームの分解速度制御システムの使われ方を整理するために「ユースケース図」を作成した（図 33)。これにより、実現するシステムの機能とともに、より詳細な振る舞いへとブレークダウンする。

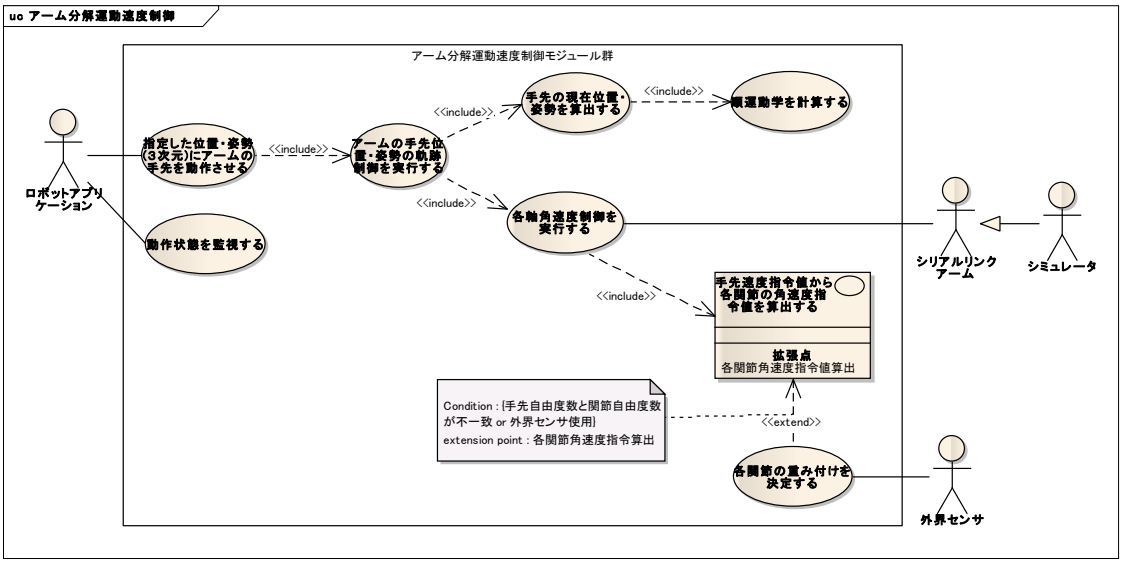

図 33 アーム分解速度制御「ユースケース図」

振る舞いを記述するためのダイアグラムとしては、アクティビティ図が利用できる。

アームの分解速度制御システムにおけるアクティビティ図は図 34 のようになる。

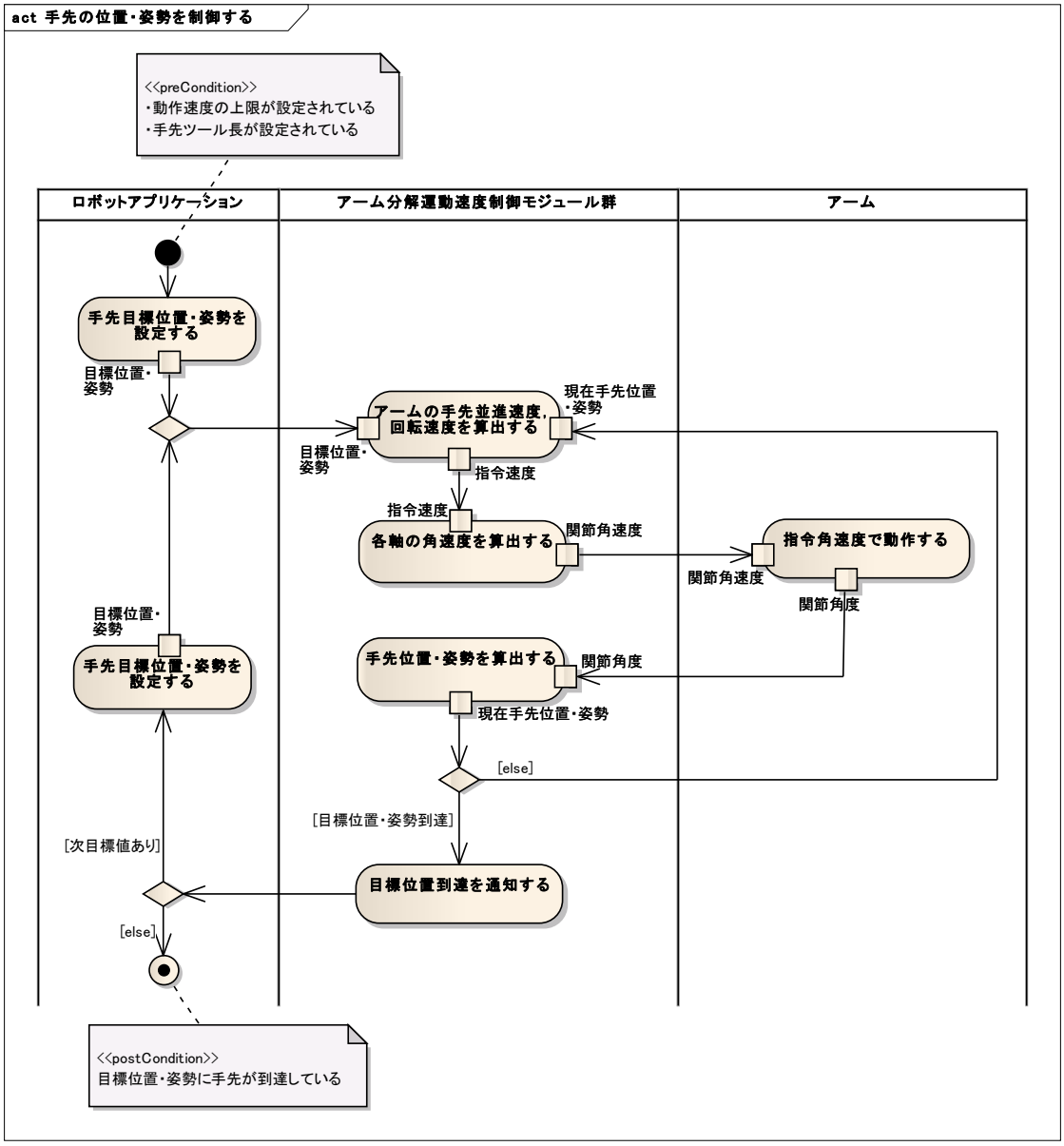

図 34 アーム分解速度制御「アクティビティ図」

以上から、アームの分解速度制御システムに必要な機能ブロックが図 35 のように定義される。

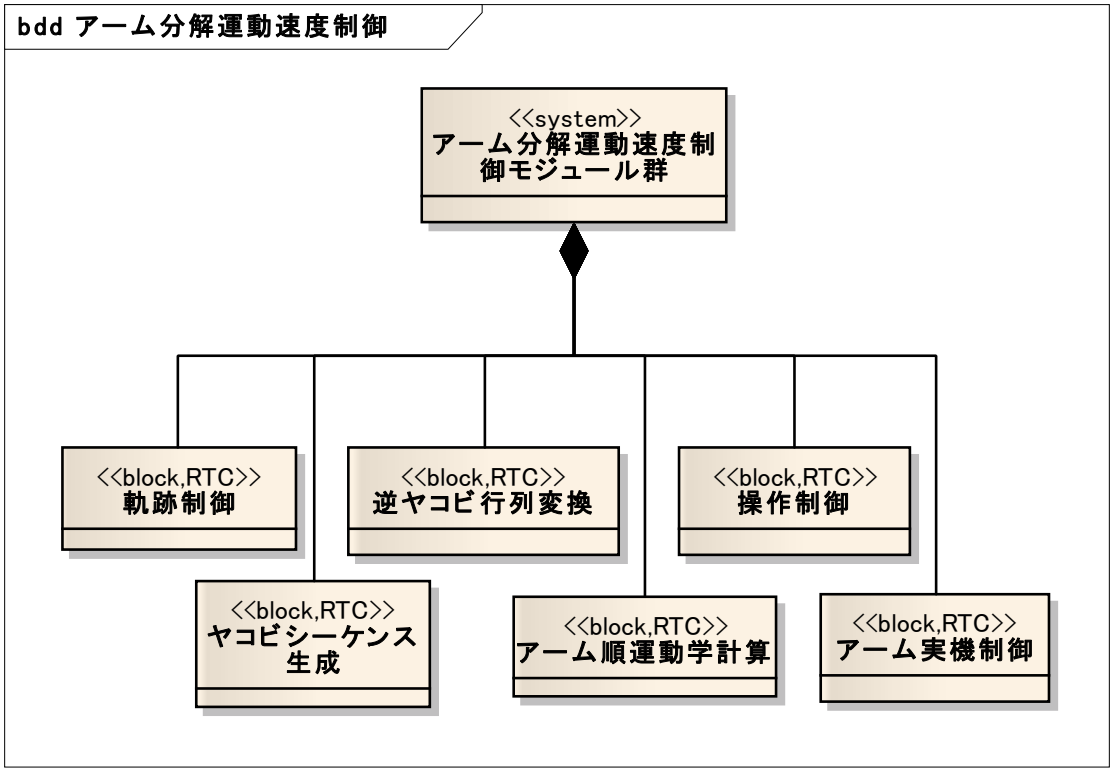

図 35 アーム分解速度制御「ブロック定義図 」

さらに、ブロック定義図により定義された機能ブロックを利用し、システム構造を記述したものは内部ブロック図と呼ばれ図 36 のように記述できる。

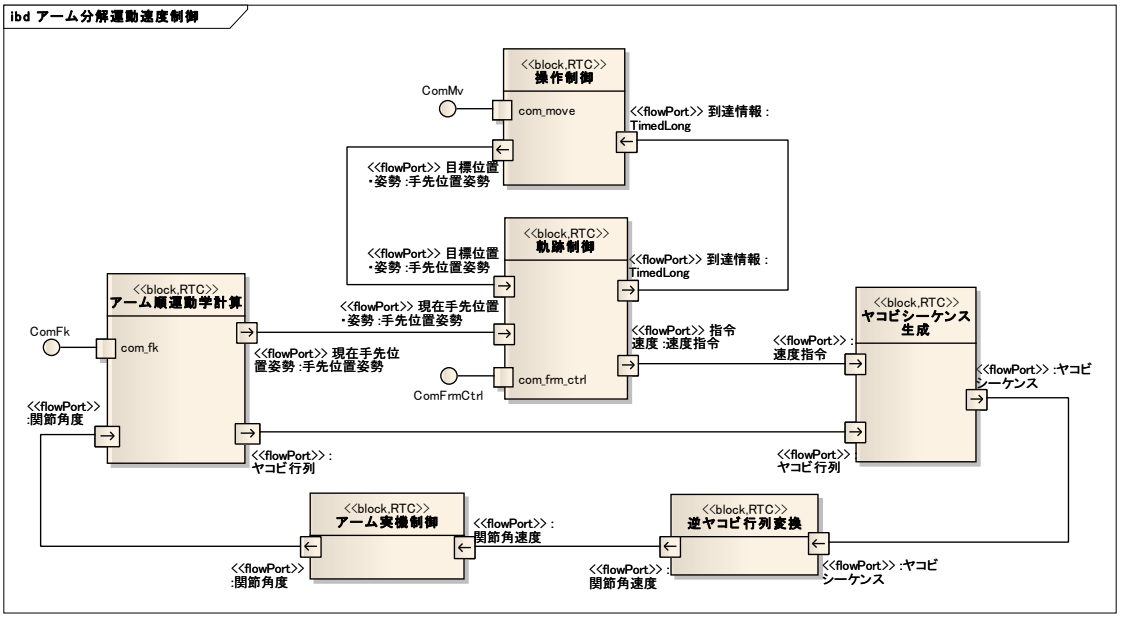

図 36 アーム分解速度制御「内部ブロック図」

以上、アームの分解速度制御システムを例にとり、ロボットシステムを SysML により表現し、システムの要求・要件定義から機能ブロックへの分解まで相互に関連

を保ちながら体系的に記述することができた。システムが SysML により記述されていれば、特定の機能がどの要件に関連付けられているかをトレースしたり、ある機能を変更した際に、どの機能ブロックに影響が及ぶかなどを容易に特定することができる。ただし、SysML で記述しただけでは、実際にトレースをすることは容易ではなく、ツールなどのサポートが必要となると考えられる。

SysML-RTCProfile 変換ツール

SysML で記述された要素のうち、ブロック定義図（bdd: Block Defintion Diagram）定義される機能要素は、RTC として実現することが可能である。したがって、bdd で定義されたブロックを上述の RTCProfile に変換することができれば、SysML によるシステム設計から RTCBuilder によるコンポーネント設計へ情報を伝達することができ、そこから RTC のソースコードを生成することができる（図 37）。コードが生成されれば、実装、デバッグなどの実装フェーズに進む OpenRTP ツールチェーンが想定したロボット設計・開発プロセスへとつなぐことができ、SysML による設計ツールをも OpenRTP ツールチェーンに組み込むことができる。

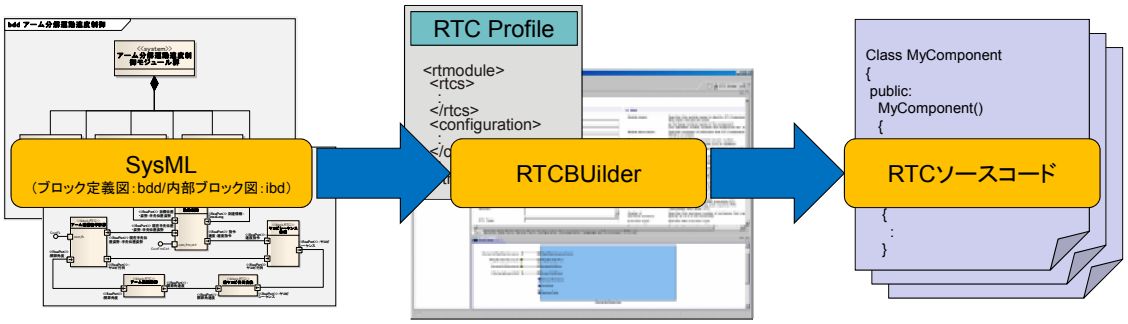

図 37 SysML-RTCProfile 変換ツール

SysML ツールとしては、Sparx Systems 社製の Enterprise Architect＋SysML プラグイン（以下 EA と呼ぶ）を想定し、EA が生成する XMI（XML Metadata Interchange）[2]を解析し RTCProfile へと変換することで実現することとした。XMI は OMG で標準化されている UML や SysML のモデルの XML による記述方式で、異なるベンダのツール間でのモデルの交換に用いられている。

RTCBuilder に SysML の XMI インポートを行う機能を追加した。図 38 のように、変換ツール起動ボタンを押し、読み込む XMI ファイルを指定する。XMI に含まれる RTCProfile へ変換可能なブロック（RTC Block）の一覧が表示されるので、変換したいブロックを指定する。これによりブロックが RTC Profile に変換され、定義された各種サービスインターフェースやデータポートなどが RTC のサービスポートやデータポートへと変換される（図 39）。

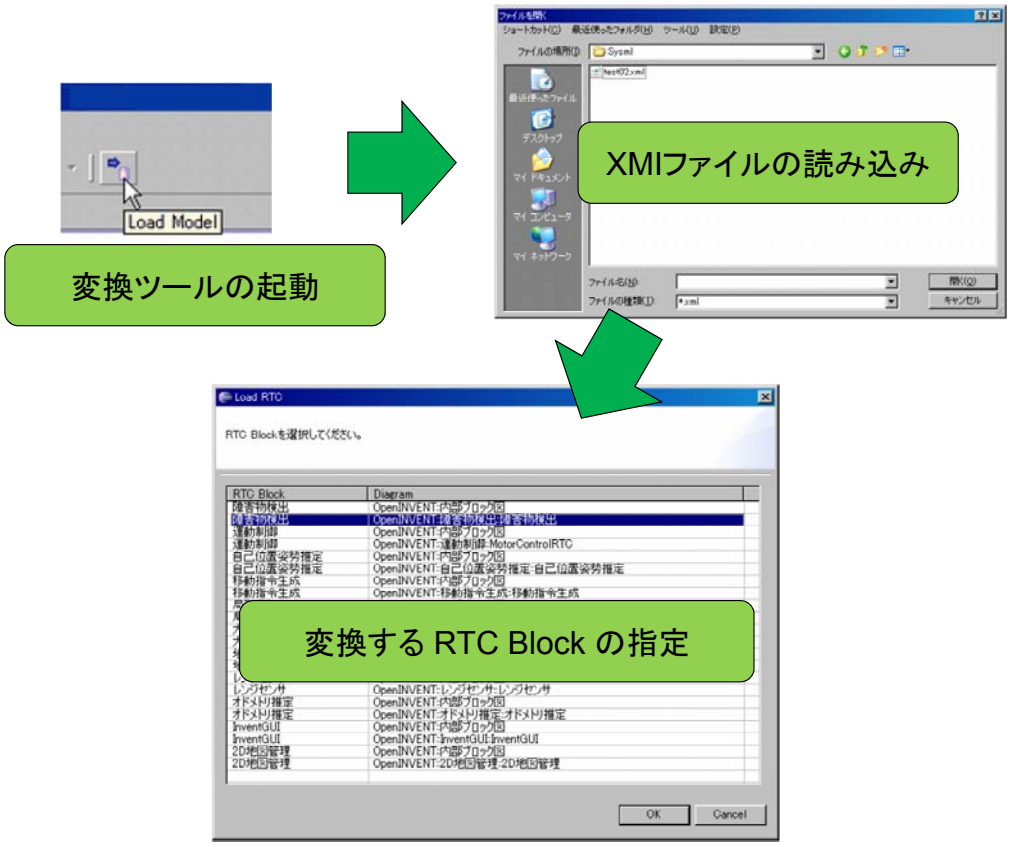

図 38 SysML-RTCProfile 変換ツールの起動

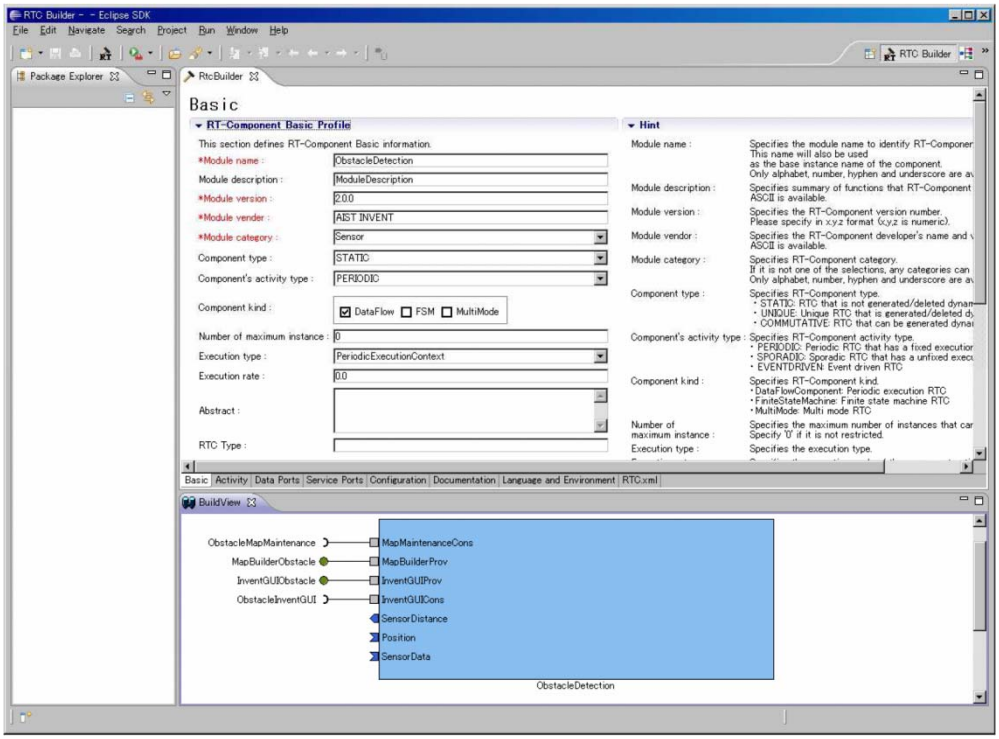

図 39 変換された機能ブロックのモデル

まとめ

ロボットシステムのサービスを中心としたシステム仕様記述について、いくつかのロボットシステムを例にとり SysML を用いてシステム記述を実施した。これにより、SysML がロボットサービス仕様記述に利用できることを確認した。

SysML により記述されたモデルのうち、機能ブロックについて、RTC Profile へ変換するツールを作成し SysML によるブロッ定義を RTC 設計に利用できることを確認した。また、このツールの実現により、ロボットシステムの設計・開発をサポートする一連の OpenRTP ツールチェーンに SysML ツールを組み込むことを実現した。システム設計の上流段階である要件定義、要求仕様記述等を体系的に記述することで、設計に対する修正や変更に対するトレーサビリティを確保する手法について道筋をつけることができた。ロボットサービス仕様記述方式としてシステムを SysML 有用であることが明らかになり、本研究開発に成果である SysML-RTCProfile 変換ツールは、RTCBuilder に統合する予定である。

本研究項目は当初の実施計画にはなかったが、より利便性の高い知能ロボットシステム構築のために実施したものである。ロボットサービスの記述に関しては、一般化するには、その記述範囲が大きく標準化にするには時期尚早であったため、OMG に標準化提案がなされているロボット・インタラクション・サービス・フレームワーク等での議論を継続して実施している。

参考文献

[1] OMG System Modeling Language (SysML™) version 1.2, formal/2010-06-01, http://www.omg.org/spec/SysML/

[2] MOF 2 XMI Mapping (XMI®), MOF XMI specification, formal/2011-08-09, SysMLhttp://www.omg.org/spec/XMI/

(b) RT コンポーネントビルダ、RT システムエディタ、RT コンポーネントデバッガ

RT コンポーネントのコード作成、デバッグ、パッケージ化等の一連の作業をシームレスに行うための RT コンポーネントビルダ、RT システムエディタ、RT コンポーネントデバッガの開発を行い、公開した。

(b-1) RT コンポーネントビルダ

RT コンポーネントビルダ（RTCBuilder）は RTC の仕様を入力することで、RTC Profile XML ファイルを生成するとともに, C++, Java, Python といった言語用の RTC のひな型コードを作成するツールである。プロジェクト開始以前には rtc-template と呼ばれるコマンドラインから利用するツールが存在したが、GUI で RTC を設計、コード生成する Eclipse 上のツールとして RTCBuilder を新たに開発

し、平成 20 年 9 月に公開した。本研究開発の最終目標は、RT コンポーネントビルダの修正・更新・機能拡張を進め、信頼性の高いツールを本プロジェクト外部に対して公開するとともに事業化を行うことである。

RTCBUilder の概要

RTCBuilder は Eclipse 上で動作するツールであり、図 40 に示すエディタ画面の各項目に RTC の名称、プロファイル情報、データポート・サービスポートに関する情報等を入力することで、上述の RTC Profile および各種言語の RTC のひな形コードを生成するツールである。生成したひな形コードに RTC の機能の中心となるロジック（コアロジック）を実装しコンパイルすることで、RTC を作成する。RTCBuilder を利用した RTC 開発の流れを図 41 に示す。

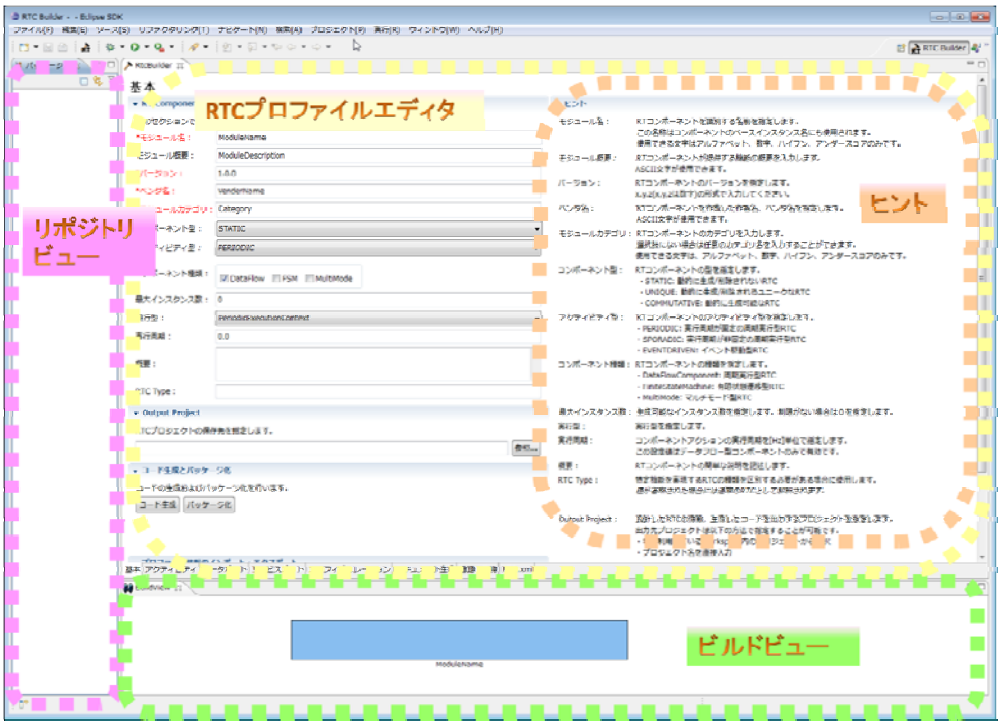

図 40 RTCBuilder 入力画面

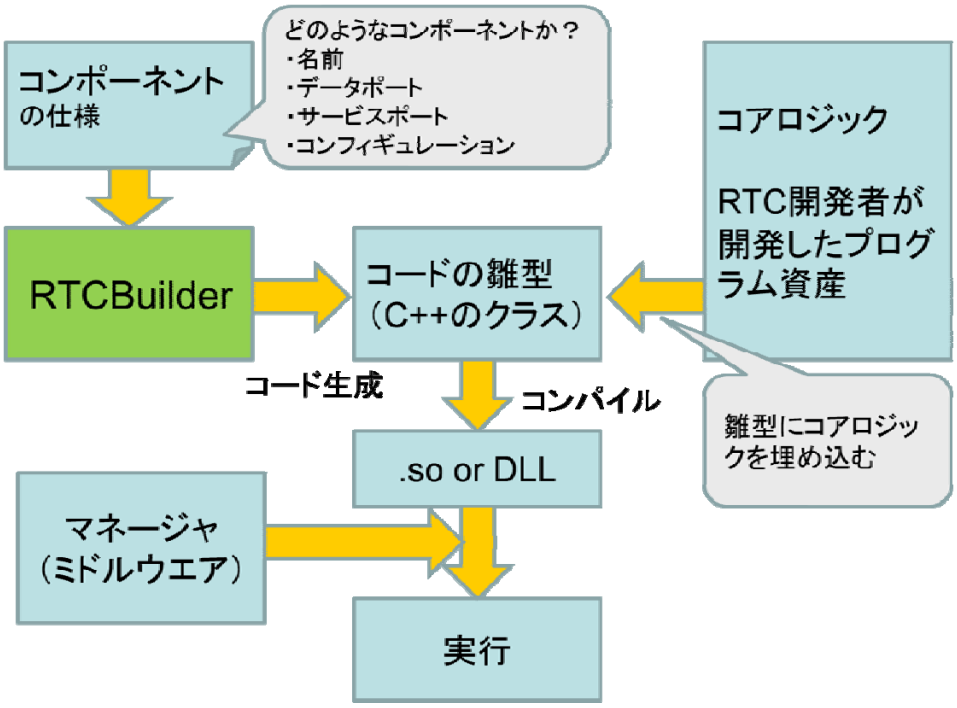

図 41 RTCBuilder を利用した RTC 開発の流れ

RTCBuilder の基本機能

RTCBuilder のエディタには、表 4 に示すサブエディタがあり、エディタ下部のタブで切り替えて使用する。

表 4 エディタの種類と機能

画面要素名	説明
基本プロファイル入力ページ	RT コンポーネントのプロファイル情報など、コンポーネントの基本情報を入力する。
アクティビティプロファイル	RT コンポーネントがサポートしているアクティビティ情報などを指定する。
データポートプロファイル	RT コンポーネントに付属するデータポートのプロファイルを入力する。
サービスポートプロファイル	RT コンポーネントに付属するサービスポートおよびサービスポートに付属するサービスインターフェースのプロファイルを入力する。
コンフィギュレーション	RT コンポーネントに設定するユーザ定義のコンフィギュレーション・パラメータセット情報およびシステムのコンフィギュレーション情報を入力する。

ドキュメント生成	生成対象の RT コンポーネントに追加する各種ドキュメント情報を入力する。
言語・環境	生成対象のコード選択や OS などの実行環境に関する情報を入力する。
RTC.xml	設定された情報を基に生成した RtcProfile の XML 形式での表示・編集を行う。

基本プロファイル

図 42 に示すように RT コンポーネントのプロファイル情報など、コンポーネントの基本情報（表 5）を入力するページである。

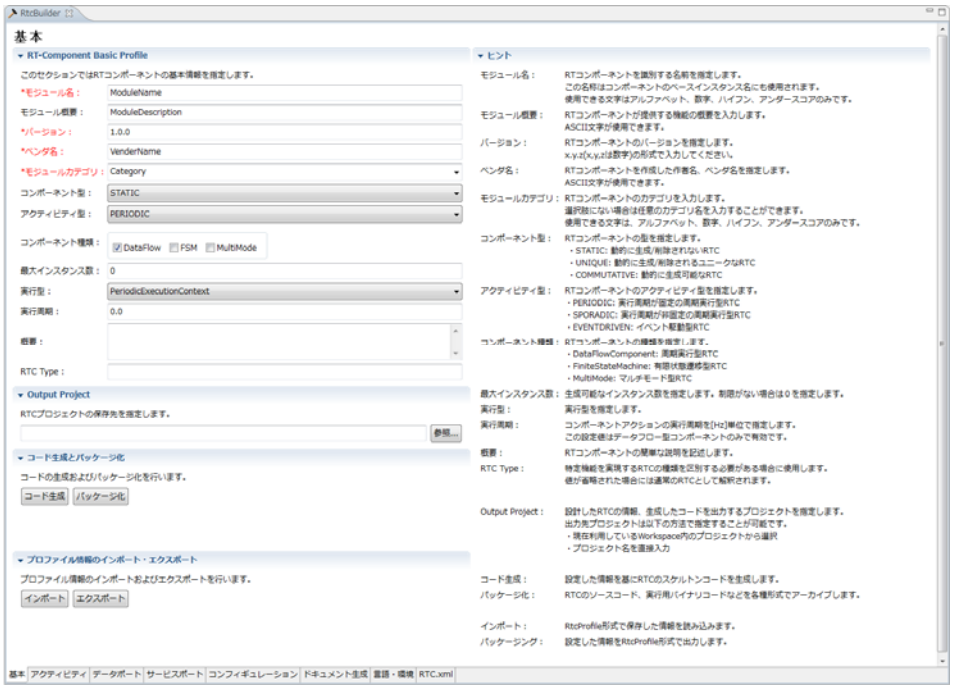

図 42 基本プロファイル入力ページ

また、このページには、入力の最後にコード生成、RTC Profile を出力する際の場所やファイル名を指定するボタン・項目等がある。

表 5 基本プロファイルの設定項目

項目	説明	必須
RT-Component Basic Profile		
モジュール名	RT コンポーネントを識別する名前です。必須入力項目。この名前は、	○

	生成されるソースコード中で、コンポーネントの名前に使用される。英数字でなければならない。	
モジュール概要	RT コンポーネントの簡単な概要説明。	-
バージョン	RT コンポーネントのバージョン。原則 x.y.z のような形式でバージョン番号を入力する。省略可能項目	-
ベンダ名	RT コンポーネントを開発したベンダ名。	○
モジュールカテゴリ	RT コンポーネントのカテゴリ。	○
コンポーネント型	RT コンポーネントの型。以下の選択肢の中から指定可能。 ・STATIC：静的に存在するタイプの RTC で、動的な生成、削除は行われないもの。 ・UNIQUE：動的に生成・削除はできるが、各コンポーネントが内部に固有状態を保持しており、必ずしも交換可能ではないタイプの RTC。 ・COMMUTATIVE：動的に生成・削除が可能で、内部の状態を持たないため、生成されたコンポーネントが交換可能なタイプの RTC。	○
アクティビティ型	RT コンポーネントのアクティビティタイプ。以下の選択肢の中から指定可能。 ・PERIODIC ：一定周期で RTC のアクションを実行するアクティビティタイプ ・SPORADIC ：RTC のアクションを不定期に実行するアクティビティタイプ ・EVENT_DRIVEN ：RTC のアクションがイベントドリブンであるアクティビティタイプ	○
コンポーネント種類	RT コンポーネントの実行形態の種類。以下の選択肢から選択可能。(複数選択肢の組み合わせ可) ・DataFlow ： 周期的にアクションを実行する実行形態 ・FSM ： 外部イベントによってアクションを実行する形態 ・MultiMode ： 複数の動作モードを持つ実行形態	○
最大インスタンス数	RT コンポーネント インスタンスの最大数。自然数を入力する。	-
実行型	ExecutionContext の型。 以下から選択可能。・PeriodicExecutionContext ： 周期実行を行う ExecutionContext・ExtTrigExecutionContext ： 外部トリガによって実行を行う	○

	ExecutionContex	
実行周期	ExecutionContext の実行周期。正の double 型の数値が入力可能（単位 Hz）。	-
概要	RT コンポーネントに関する説明。	ー
RTC Type	特定機能を実現する RT コンポーネントを区別する必要がある場合に指定。	ー
Output Project		
	生成コードの出力対象プロジェクト名。設定したプロジェクトが存在する場合には、そのプロジェクト内に、設定したプロジェクトが存在しない場合には、新規プロジェクトを生成する。	○

アクティビティプロファイル

図 43 に示すように、作成する RTC において、どのコールバックを実装するか、またその実装される予定のロジックについてのドキュメントを入力するページである。アクティビティとは RTC のライフサイクルの状態遷移ごとに定められたコールバック関数群のことであり、OMG RTC 仕様では ComponentAction とも呼ばれる。

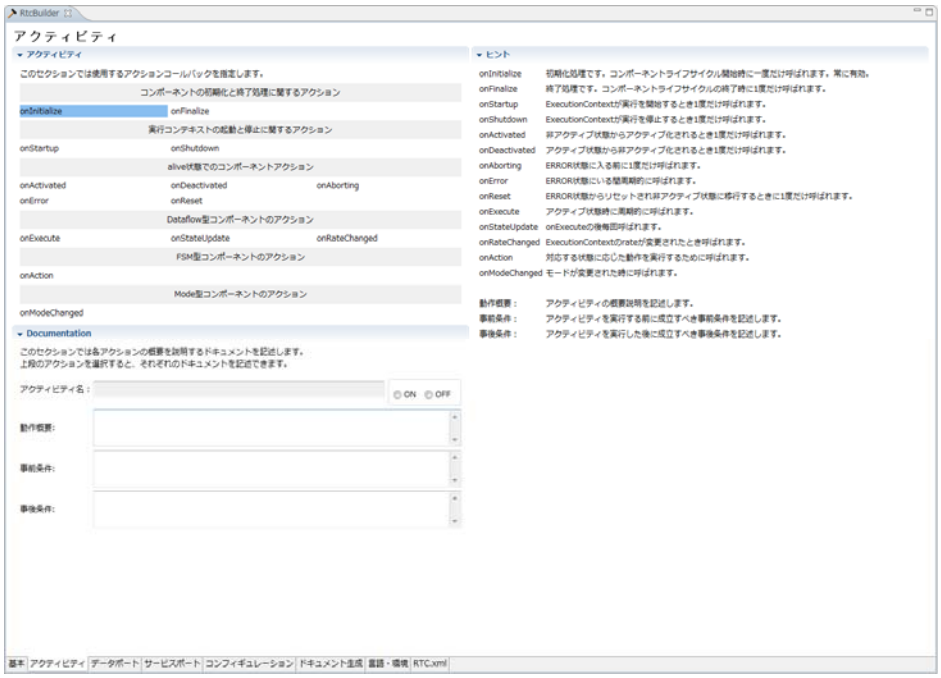

図 43 アクティビティプロファイル入力画面

アクティビティプロファイルの各入力項目の種類と意味は表 6 のとおりである。

表 6 アクティビティプロファイルの設定項目

項目	説明	必須
Activity Profile		
onInitialize	初期化処理です。コンポーネントライフサイクルの開始時に一度だけ呼ばれる。	-
onFinalize	終了処理です。コンポーネントライフサイクルの終了時に 1 度だけ呼ばれる。	ー
onStartup	ExecutionContext が実行を開始するとき 1 度だけ呼ばれる。	ー
onShutdown	ExecutionContext が実行を停止するとき 1 度だけ呼ばれる。	ー
onActivated	非アクティブ状態からアクティブ化されるとき 1 度だけ呼ばれる。	ー
onDeactivated	アクティブ状態から非アクティブ化されるとき 1 度だけ呼ばれる。	ー
onAborting	ERROR 状態に入る前に 1 度だけ呼ばれる。	ー
onError	ERROR 状態にいる間に呼ばれる。	ー
onReset	ERROR 状態からリセットされ非アクティブ状態に移行するときに 1 度だけ呼ばれる。	ー
onExecute	アクティブ状態時に周期的に呼ばれる。	ー
onStateUpdate	on_execute の後毎回呼ばれる。	ー
onRateChanged	ExecutionContext の rate が変更されたとき呼ばれる。	ー
onAction	対応する状態に応じた動作を実行するために呼ばれる。	ー
onModeChanged	モードが変更された時に呼ばれる。	ー
Documentation		
アクティビティ名	現在選択されているアクティビティの名称を表示する。	ー
動作概要	対象アクティビティが実行する動作の概要説明を記述する。	ー
事前条件	対象アクティビティを実行する前に成立すべき事前条件を記述する。	ー
事後条件	対象アクティビティを実行した後に成立する事後条件を記述する。ただし、事前条件が満たされない状態で対象アクティビティが実行された場合は事後条件の成立は保証されない。	ー

データポートプロファイル

RT にはデータストリームを送受信するためのポート：データポートを付加することができる。RT コンポーネントは 0 個以上のデータポートを持ち、事前に定義されたデータ型に加えて、独自のデータ型を IDL で定義して利用することができる。データポートプロファイルページ（図 44）では、データポートの名前、型等を入力しデータポートを定義する。

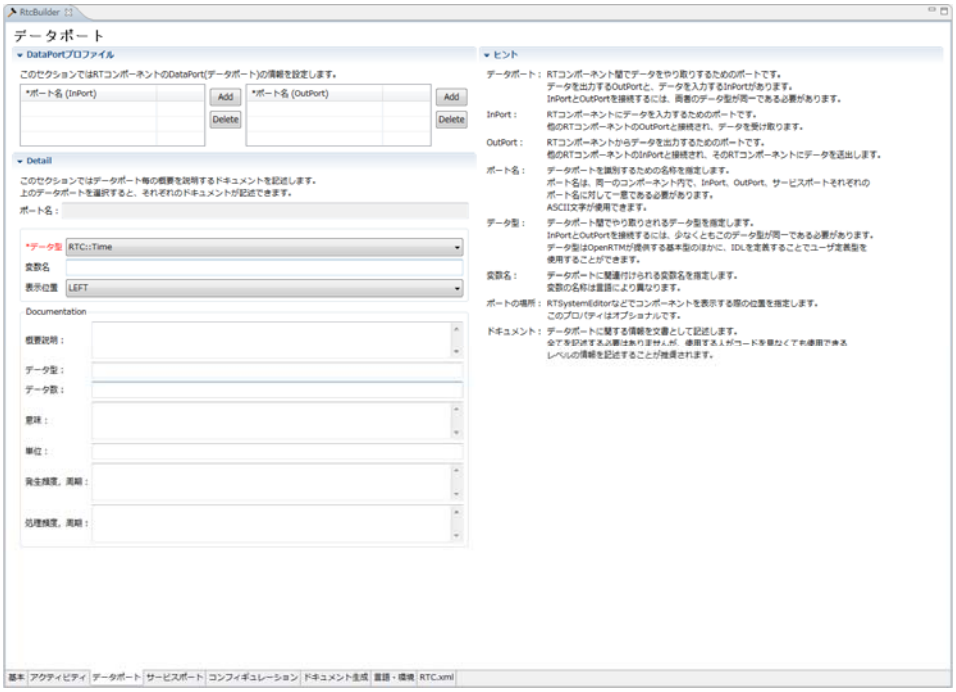

図 44 データポートプロファイル入力画面

各ポート（InPort/OutPort）を新規に追加する場合は、各セクションの「 Add 」ボタンを押す。次に使用するデータ型をプルダウンメニューから選択する。この他、このデータポートがどういった意味のデータを交換するかなどをドキュメント項目として入力する。入力項目は表 7 のとおりである。

表 7 データポートプロファイルの設定項目

項目	説明	必須
DataPort プロファイル		
ポート名	DataPort の名称です。半角英数字のみ入力可能。 Data OutPort、Service Port と併せてポート名称は一意でなければならない。	○
Detail		

ポート名	現在選択されている Data Port を「ポート名(InPort/OutPort)」の形式で表示。	ー
データ型	DataPort が扱うデータ型。 設定画面にて指定した IDL 内で定義されているデータ型が利用可能。	○
変数名	DataPort に対応する変数名。	ー
表示位置	ビルドビュー内での Data InPort の表示位置。	○
概要説明	データポートに対する概要説明を記述する。	ー
データ型	データポートの扱う型に対する説明を記述する。	ー
データ数	データが配列になる場合など、データ数に関する説明を記述する。	ー
意味	データの意味の説明を記述する。	ー
単位	データ単位に関する説明を記述する。	ー
発生頻度，周期	データの発生頻度、周期に関する説明を記述する。	ー
処理速度，周期	データの処理速度、処理周期に関する説明を記述する。	ー

サービスポートプロファイル

データポートがデータストリームを送受信するためのポートであるのに対し、サービスポートはコマンドやオペレーションといった単位で RTC 間がコミュニケーションを行うためのチャネルである。サービスポートの情報を入力するページは図 45 のとおり。

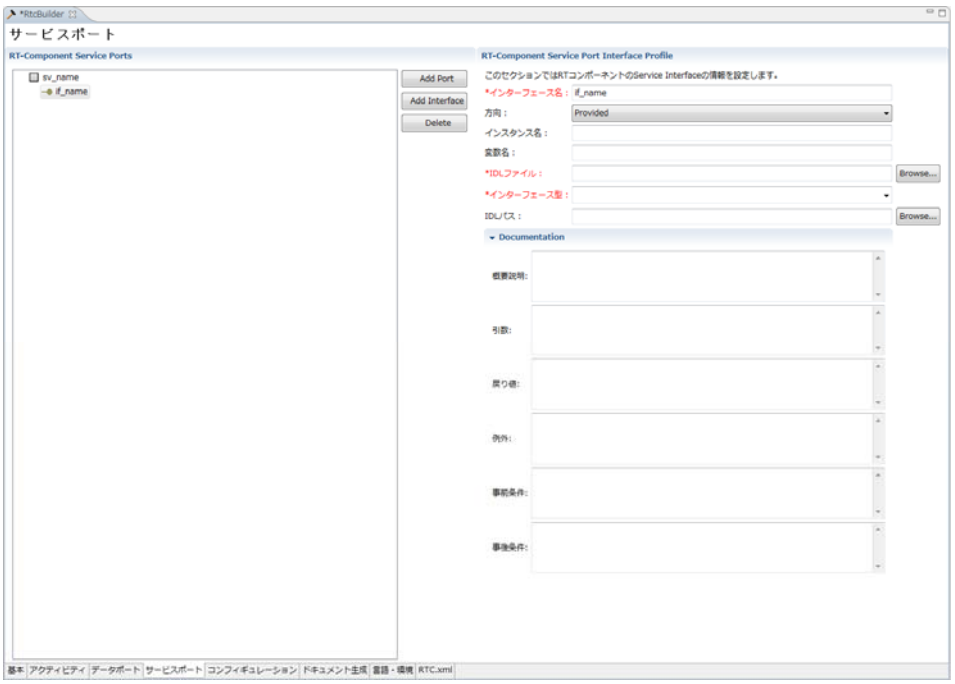

図 45 サービスポートプロファイル入力ページ

新規サービスポートは、画面左側「 RT-Component Service Ports 」欄の「 Add Port 」を選択することで追加する。画面左側「 RT-Component Service Ports 」にてサービスポートを選択した状態で、「Add Interface 」を選択することで新規サービスインターフェースを追加することができる。画面左側「 RT-Component Service Ports 」にて、サービスポートもしくはサービスインターフェースを選択した状態で「 Delete 」を選択することで、選択したポート／インタフェースを削除する。以下に各入力項目について述べる。

表 8 サービスポートの設定項目

項目	説明	必須
RT-Component Service Port Profile		
ポート名	サービスポートの名称です。半角英数字のみ入力可能。 Data InPort、Data OutPort、Service Port 名称は一意でなければならない。	○
表示位置	ビルドビュー内でのサービスポートの表示位置。	○
Documentation		
概要説明	サービスポートに対する概要説明を記述する。	ー

I/F 概要説明	サービスポートに付属するサービスインターフェースの概要説明を記述する。	ー

表 9 サービスポートインターフェースの設定項目

項目	説明	必須
RT-Component Service Port Interface Profile		
インタフェース名	サービスインターフェースの名称です。半角英数字のみ入力可能。 サービスインターフェース名は重複不可。	○
方向	サービスインターフェースの種類。以下の選択肢から選択可能。 ・Provided：提供インタフェース(Service Provider 用) ・Required：要求インタフェース(Service Consumer 用)	○
インスタンス名	サービスインターフェースのインスタンス名。半角英数字のみ入力可能。	○
変数名	サービスインターフェースの変数名。省略された場合は、インスタンス名を使用する。	ー
IDL ファイル	サービスインターフェースで使用する IDL ファイル名を指定する。 「Browse...」ボタンをクリックすると、ファイル選択ダイアログが表示される。	○
インタフェース型	サービスインターフェースで使用するサービスの型。IDL file を指定すると IDL 内で定義されている型情報が表示される。半角英数字のみ入力可能。	○
IDL Path	IDL のサーチパス。「Browse...」ボタンをクリックすると、ディレクトリ選択ダイアログが表示される。	ー
Documentation		
概要説明	サービスインターフェースに対する概要説明を記述する。	ー
引数	サービスインターフェースの引数に関する説明を記述する。	ー
戻り値	サービスインターフェースの戻り値に関する説明を記述する。	ー
例外	サービスインターフェースの例外に関する説明を記述する。	ー
事前条件	サービスインターフェースのオペレーションを実行前に満たしておくべき事前条件に関する説明を記述する。	ー

事後条件	サービスインターフェースのオペレーションを実行後に満たす事後条件に関する説明を記述する。	ー

コンフィギュレーション

同一の RTC を様々なシステムに適用する際に、RTC 内部のロジックで使用する各種パラメータを管理、実行時に外部から設定・変更する機能がコンフィギュレーション機能である。このような RT コンポーネントに設定するユーザ定義のコンフィギュレーション・パラメータ情報およびその他システムのコンフィギュレーション情報を入力するページを図 46 に示す。

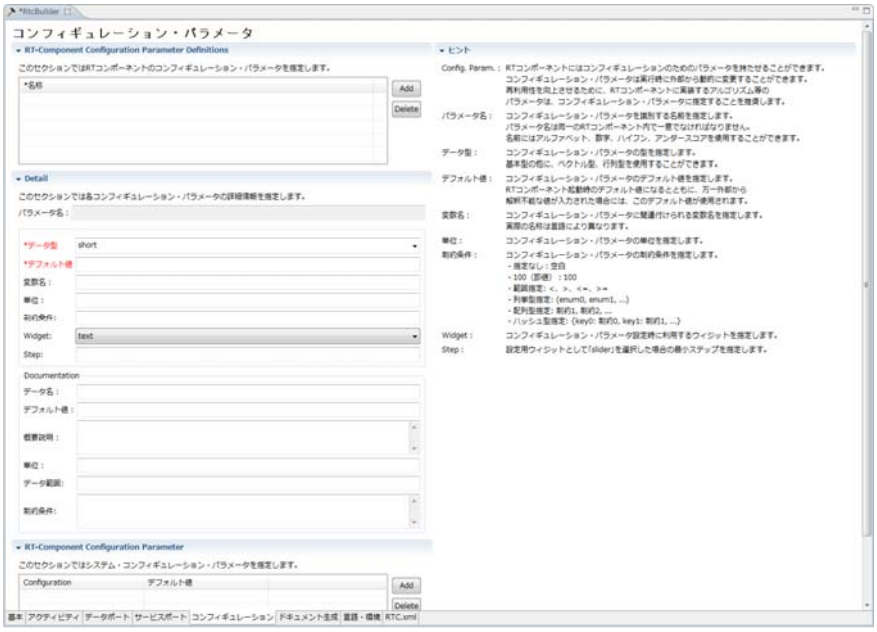

図 46 コンフィギュレーション入力ページ

ユーザ定義コンフィギュレーション・パラメータ情報およびシステム・コンフィギュレーション情報を新規に追加する場合は、「Add」ボタンを押し、名称、データ型等必要な項目を入力する。このように入力されたコンフィギュレーション・パラメータは、ソースコード内に専用の変数として宣言され、プログラム内部で代入・参照することができると共に、RTSystemEditor 等を用いて 外部から参照・変更することができる。

以下に各入力項目について説明する。

表 10 コンフィギュレーション・パラメータの設定項目

項目	説明	必須
RT-Component Configuration Parameter Definitions		
名称	ユーザ定義コンフィギュレーション・パラメータの名称です。半角英数字のみ入力可能。ユーザ定義コンフィギュレーション・パラメータ名称は一意でなければならない。	○
Detail		
パラメータ名	現在選択されているユーザ定義コンフィギュレーション・パラメータを表示する。	―
データ型	ユーザ定義コンフィギュレーション・パラメータのデータ型。設定画面にて指定した IDL 内で定義されているデータ型が利用可能。	○
デフォルト値	ユーザ定義コンフィギュレーション・パラメータのデフォルト値。2バイト文字を含む任意の値を設定可能。	○
変数名	ユーザ定義コンフィギュレーション・パラメータの変数名。半角英数字のみ入力可能。	―
単位	ユーザ定義コンフィギュレーション・パラメータの単位。	―
制約条件	ユーザ定義コンフィギュレーション・パラメータに対する制約条件を記述する。制約条件の記述方法については、表 11 制約情報の記述方式を参照のこと。	―
Widget	RTSystemEditor の ConfigurationView にてコンフィギュレーション・パラメータを設定する際に利用するコントロールを指定する。以下の値から選択可能。 ・text：テキストボックス(デフォルト設定) ・slider：スライダ ・spin：スピンボタン ・radio：ラジオボタン	○
Step	入力用コントロールとして、「slider」を選択した場合に、スライダの刻み幅を指定する。	―
パラメータ名	現在選択されているユーザ定義コンフィギュレーション・パラメータを表示する。	―
データ名	ユーザ定義コンフィギュレーション・パラメータの名称に対する説明を記述する。	―

デフォルト値	ユーザ定義コンフィギュレーション・パラメータのデフォルト値に対する説明を記述する。	ー
概要説明	ユーザ定義コンフィギュレーション・パラメータに対する概要説明を記述する。	ー
単位	ユーザ定義コンフィギュレーション・パラメータの単位に対する説明を記述する。	ー
データレンジ	ユーザ定義コンフィギュレーション・パラメータのデータ範囲に関する説明を記述する。	ー
制約条件	ユーザ定義コンフィギュレーション・パラメータの制約条件に関する説明を記述する。	ー
RT-Component Configuration Parameter		
Configuration	設定を行うコンフィギュレーション名。一覧から選択する。	○
デフォルト値	設定対象コンフィギュレーションのデフォルト値。予めデフォルト値が設定されている項目については、名称選択時にデフォルト値が設定される。	ー

表 11 制約条件の書式

設定内容	設定書式
指定なし	空白
100(即値)	100
100 以上	x >= 100
100 以下	x<=100
100 超	x>100
100 未満	x<100
100 以上 200 以下	100<=x<=200
100 超 200 未満	100<x<200
列挙型	(9600,19200,115200)
配列型	x>100, x>200, x>300
ハッシュ型	{key0: 100<x<200, key1: x>=100}

ドキュメント生成

図 47 に示すように、コンポーネントの概要情報、入出力に関する情報、アルゴリズムなどの振る舞いに、ついてドキュメントを入力するページである。入力項目を表 12 に示す。

図 47 ドキュメント入力ページ

このページで入力された情報は、生成されたコードに Doxygen 形式で埋め込まれる。ソースコードを Doxygen で処理することで、LaTeX や HTML 形式でドキュメントを出力することができる（図 48）。

表 12 ドキュメント設定項目

項目	説明	必須
コンポーネント概要		
概要説明	生成する RT コンポーネントの概要説明を記述します。	ー
入出力	RT コンポーネントの入出力に関する概略説明を記述します。	ー
アルゴリズムなど	RT コンポーネントが使用しているアルゴリズムなどの説明を記述します。	ー
その他		
作成者・連絡先	RT コンポーネントの作成者および連絡先に関する情報を記述します。	ー
ライセンス，使用条件	RT コンポーネントのライセンス情報、使用条件に関する情報を記述します。	ー

参考文献	参考文献情報を記述します。	ー
バージョンアップログ		
VersionUp Log	今回の変更内容に関するログ情報を記述します。	ー
ライセンス，使用条件	過去のバージョンアップ時のログ情報を表示します。	ー

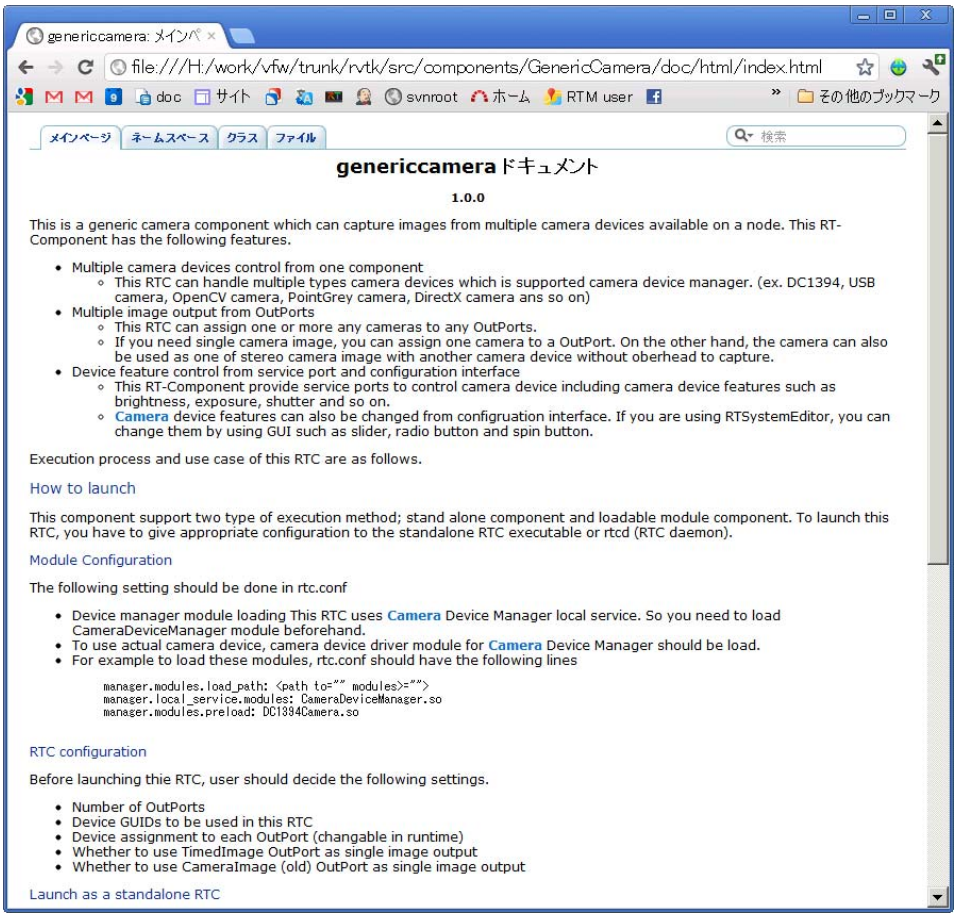

図 48 生成されたドキュメント（HTML）の例

言語・環境

図 49 に示すように、入力した RT コンポーネント仕様に基づき生成するテンプレート・ソースコードの言語選択や、OS 等の実行環境、依存ライブラリなどを入力するページである。

図 49 言語・環境ページ

生成したい言語のセクションを選択し、各言語固有の設定情報を入力する。コード生成実行時（基本プロファイル入力ページの「コード生成 」ボタン押下時）に選択されていたセクションの言語用テンプレートコードが生成される。 設定可能な項目を表 13 に示す。

表 13 言語・環境情報入力ページ設定項目

項目	説明	必須
言語	生成対象の言語を指定します。	○
環境		
Version	生成対象 RTC を実装している言語のバージョン情報を設定します。	―
OS	生成対象 RTC が動作する OS 情報を設定します。	―
OS Version	生成対象 RTC が動作する OS のバージョン情報を設定します。	―
CPU	生成対象 RTC が動作する CPU アーキテクチャ情報を設定します。	―
その他 OS 情報	生成対象 RTC が動作する OS について、バージョン情報以外の補足情報を設定します。	―

その他 CPU 情報	生成対象 RTC が動作する CPU について、アーキテクチャ情報以外の補足情報を設定します。	ー
ライブラリ情報		
Name	生成対象 RTC が利用する外部ライブラリの名称を指定します。	○
Version	生成対象 RTC が利用する外部ライブラリのバージョン情報を指定します。	ー
Info.	生成対象 RTC が利用する外部ライブラリの補足情報を指定します。	ー

RTC.xml

上記の各設定項目は上述の RTC Profile に準拠した仕様記述方式の XML で表現される。通常この XML ファイルをユーザが直接編集する必要はないが、内容の確認および特別に編集が必要な場合のため、XML ファイルエディタが提供されている。設定項目を編集し、「Update」ボタンを押すと現在の設定内容が XML 形式で表示され、同時に編集も行うこともできる。

図 50 RTC.xml XML エディタ

OpenRTM-aist の新バージョンへの対応

プロジェクト期間中、OpenRTM-aist はバージョン 0.4、1.0 および 1.1-RC 版へとバージョンアップした。0.4 から 1.0 へのバージョンアップにおいては、API の変更があったため、ひな形コードにも大幅な変更が行われた。また、バージョン 0.4 配布後にユーザから寄せられたフィードバックを元に、入力画面の大幅な刷新を行った。まずひとつに、バージョン 0.4 においては、図 51 の左側の入力テキストボッ

クスのみの画面構成であったが、入力項目が多い上にそれぞれの入力項目の意味する内容が初心者にはわかりづらかった。このため、図 51 のエディタ画面右側のように、各項目の意味と入力例などをヒントとして表示するようにした。これにより、マニュアルなどを参照しなくても、各項目を素早く入力できるようになった。また、データポートの入力に関しても、事前に登録されたディレクトリから IDL を読み込み、利用可能なデータ型をプルダウンから選択可能にするなど、入力の効率化と入力ミス削減のために様々な GUI 上の工夫を施した。

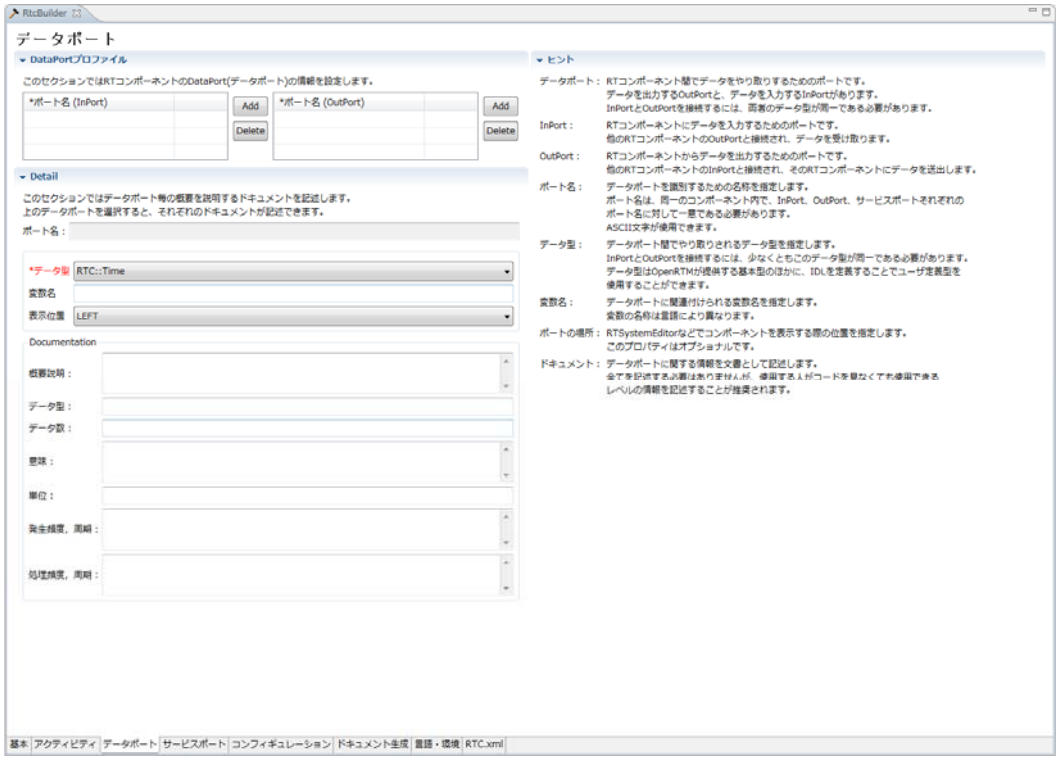

図 51 エディタ画面とヒント表示

また、作成する RTC は、第 3 者が利用する際に中身をブラックボックスとしても提供できるようにすべきであり、このため RTCBuilder にドキュメント入力機能も追加された。入力されたドキュメントは、ソースコード内に埋め込まれ、Doxygen で処理することで、LaTeX や HTML などに変換することが可能である。

また、作成する RTC の仕様は実装開始後も度々変更されるケースがあるため、実装開始後も RTCBuilder に戻り仕様変更を行った上で、再度コード生成ができるような機能を実現した。生成コードはある特定のタグで囲み、コードジェネレータはタグ内にしかコードを上書きしないため、開発者が作成したコードをそのままのこした上で、自動生成コードを再生成することができる。また、コードの上書きをする場合、図 52 に示すように生成済みコードと生成中コードの差分を表示する機能も

追加し、新旧のコードを見ながら新たなコード生成を行うことが可能となった。

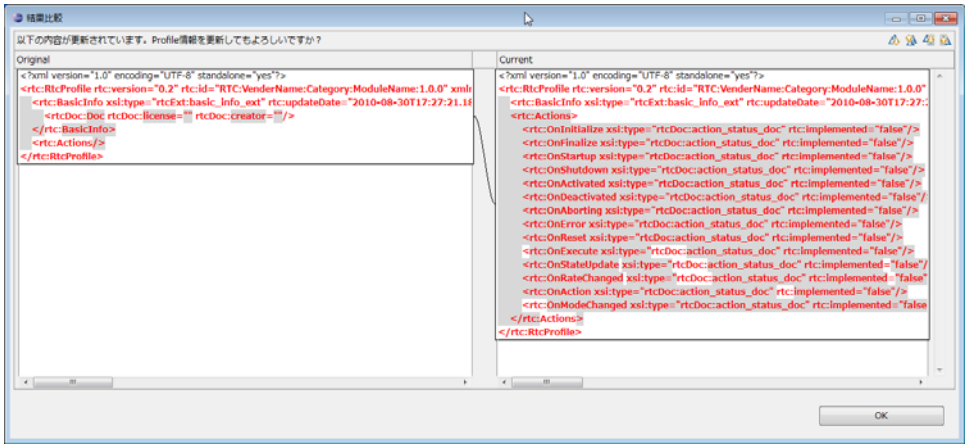

図 52 既存コードと生成コードの差分表示例

CMake への対応

近年、マルチプラットフォーム用のビルドシステムとして CMake が多くのソフトウェアプロジェクトで利用されている。CMake は一種のメタ make コマンドで、CMakeList.txt と呼ばれる CMake 用汎用 Makefile を作成することで、一般的な Makefile や Eclipse の project ファイルに加えて、Visual C++ （Visual Studio）等のためのソリューションファイル、プロジェクトファイルを生成することが可能である（図 53、図 54）。RTCBUilder および rtc-template は独自のツールにより Visual C++ 2005/2008 のソリューション・プロジェクトファイルを生成する機能を持っていたが、Visual C++ 2010 以降これらのファイルフォーマットが大幅に変更されたため、今後のメンテナンスの容易さ等を考慮し CMake へ移行することとした。

CMake にはパッケージ作成機能もあるため、これを利用することで、作成したコンポーネントパッケージ化も容易になった（図 55）。また、RTCBuilder が生成する CMakeList.txt はドキュメントも同時に生成できるように設定されており、RTCBuilder のドキュメント入力ページで入力した内容からドキュメントも同時に生成できるようになっている（図 56）。

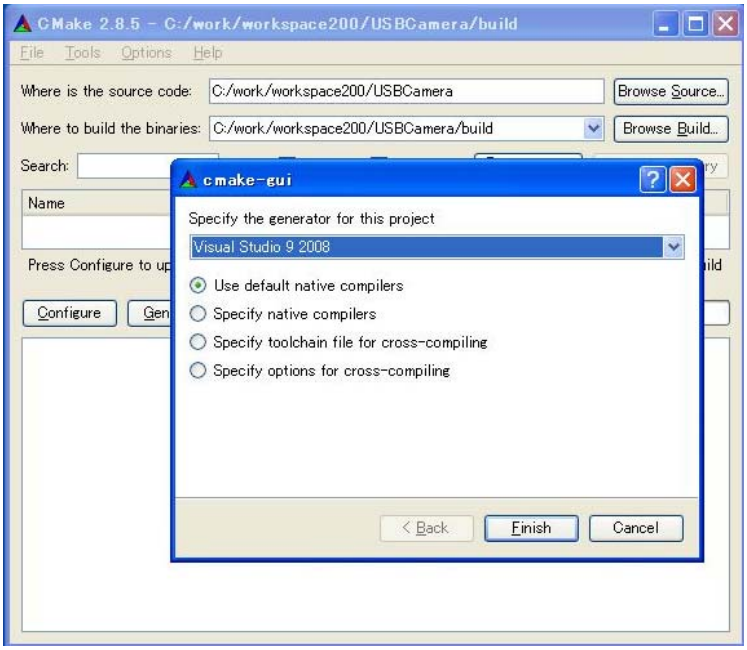

図 53 CMake による生成するプロジェクトタイプの指定

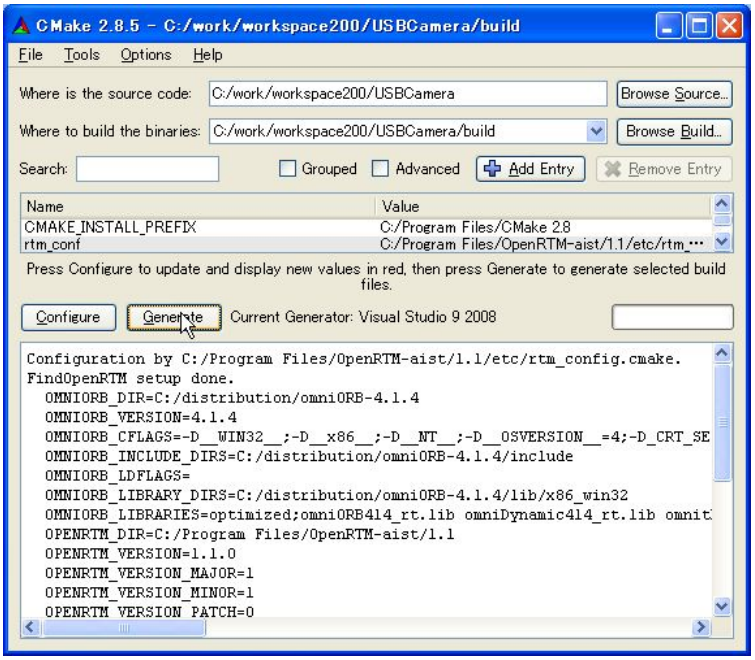

図 54 CMake による Visual Studio のプロジェクトファイルの生成

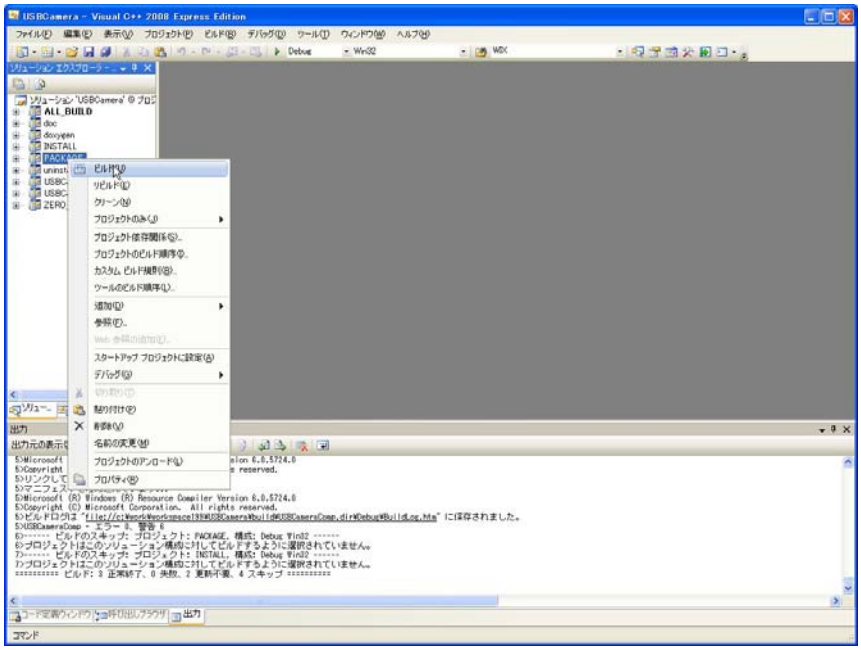

図 55 CMake を利用した Visual Studio からのパッケージ作成

図 56 Doxygen により生成されたドキュメントの例

リリース履歴

- 平成 20 年 9 月 8 日：RTCBUilder・RTSystemEditor version 0.4 リリース
- 平成 22 年 2 月 9 日：RTCBUilder・RTSystemEditor version 1.0-RC1 リリース
- 平成 22 年 6 月 1 日：RTCBUilder・RTSystemEditor version 1.0-RELEASE リリース
- 平成 23 年 5 月 2 日：RTCBUilder・RTSystemEditor version 1.1-RC1 リリース

まとめ

以上、RTC のひな形コードを自動生成する RTCBuilder の機能について説明した。本プロジェクトで作成したこのツールは、14000 回以上ダウンロードされ、プロジェクト参加組織のみならず一般ユーザにも広く使われ、RTC 作成のための標準ツールとして定着したといえる。本ツールは、RT コンポーネント開発における基本ツールであるため、オープンソースラインセンスと個別ライセンスのデュアルライセンスで公開を行っている。現在は、サポートに関しても開発者、ユーザによるコミュニティを中心に継続的な体制を構築しているが、商業利用も可能なライセンス形態をとっている。

(b-1) RT コンポーネントデバッガ

RT システムに、RT コンポーネントを組み込むためには、RT コンポーネント単体で事前に十分なテストとデバッグを行う必要がある。本項目では、RT コンポーネント単体のデバッグを行うためのツールとして、RT コンポーネントデバッガの研究開発を行う。最終目標は、RT コンポーネントデバッガに対して、機能・使い勝手向上、バグフィックスを行い、信頼性の高いツールを本プロジェクト外部に対して公開するとともに、事業化を行うことである。

成果の概要

RT システムに、RT コンポーネントを組み込むためには、RT コンポーネント単体で事前に十分なテストとデバッグを行う必要がある。しかしながら、これまでは最適なテストツールがなく、各々の開発者がテスト用のドライバやスタブを作成しており、RTC の開発効率は良いとは言えなかった。そこで我々は、RTC のテストやデバッグを行うツールとして RTC デバッガを開発した。実際に RTC デバッガを RTC 開発に適用したところ、RTC 単体のテスト工数を 4 割程度削減でき、開発効率が向上することを確認した。

RT コンポーネントデバッガは、RT コンポーネントのデバッグに特化した以下の機能を備える。

- データ入力ポート検証機能
- データ出力ポート検証機能
- アクティビティ検証機能
- コンフィグレーション検証機能
- サービスポート検証機能
- RT コンポーネント実行コンテキスト制御機能
- データプロット機能
- データストア機能

- データ再生機能

さらに、RT コンポーネントデバッガは、RT コンポーネントビルダや RT システムエディタなどの各種ツールと容易に連携できるよう、Eclipse プラグイン版および、Eclipse-RCP 版の 2 つの実装を開発した。

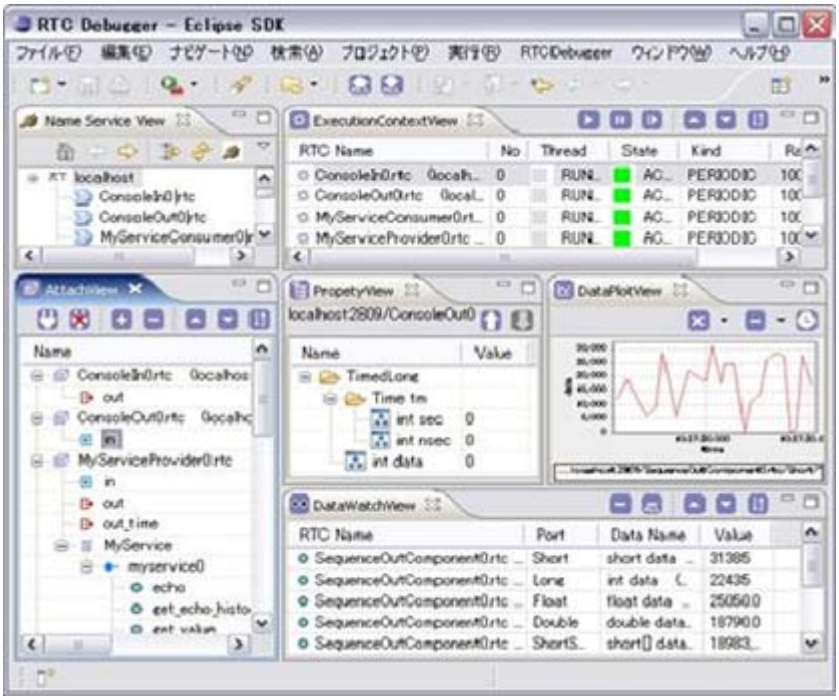

図 57　RTC デバッガ

以降に、RTC デバッガが備える主な機能を示す。
データ入出力／サービスポート検証機能では、データポートへのデータ書き込みや、出力データの表示、サービスポートの提供するサービスを呼び出すことができる。

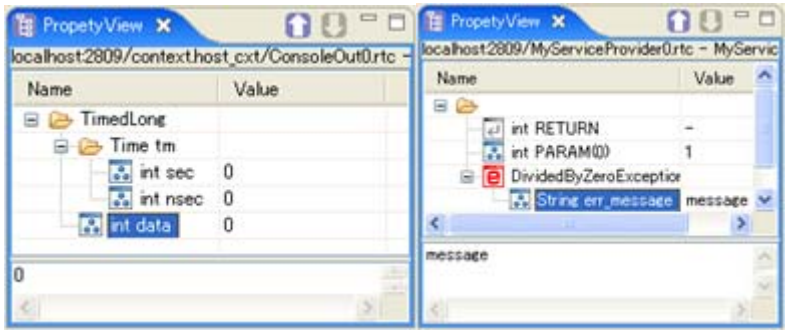

図 58 データ入出力／サービスポート検証機能

実行コンテキスト検証機能では、RT コンポーネントの状態を監視や RT コンポーネントへのコマンド送信や、動作周期の変更を行うことができる。

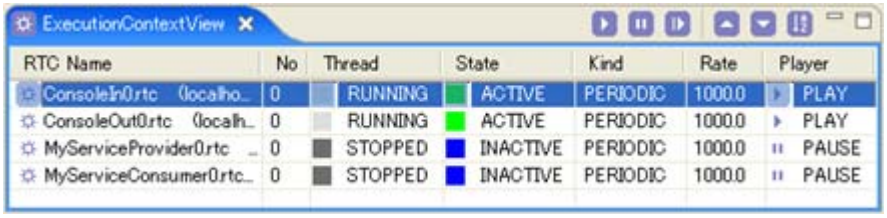

図 59　実行コンテキスト検証機能

データプロット・ストア・再生機能では、データポートから出力されたデータのグラフ化や画像表示を行うことができる。さらに、データポートの出力をファイルにエクスポートする機能や、ファイルからインポートしたデータをデータポートに入力する機能を持つ。

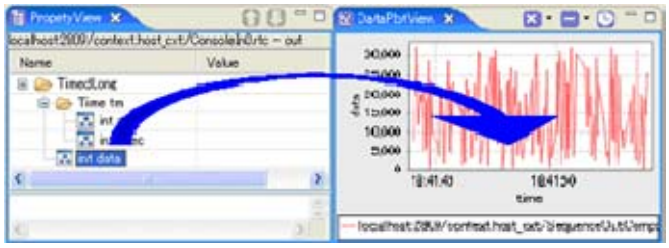

図 60　データプロット機能（グラフ表示）

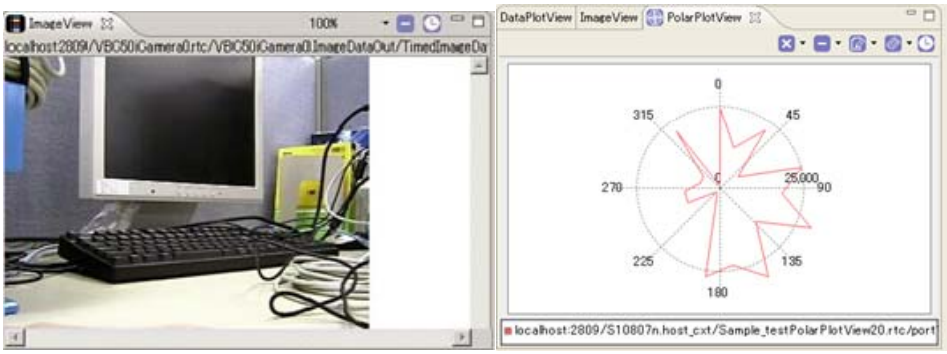

図 61　データプロット機能（画像表示・極座標表示）

まとめ

RT コンポーネントデバッガのソフトウェアとマニュアルは、RT ミドルウェアによるロボットシステムの開発効率を向上するためのツールとして、セックのロボットサイト（http://www.sec.co.jp/robot/index.html）にて無償公開している。NEDO 次

世代ロボット知能化技術開発プロジェクトの参画機関を始め、その他の大学や企業でも利用されており、RT ミドルウェアの普及に貢献している。

(b-3) RT システムエディタ

RT コンポーネントを組み合わせてシステムの設計・開発を行う際には、静的または動的なシステム構成や、構成したシステムのシミュレーションといった作業が想定される。 静的システム構成とは、コンポーネント間の静的接続を構成する作業であり、動的システム構成とは、時間軸及びイベントによりシステム構成を動的に切り替え一連のシナリオを実行するシーケンスを作成する作業である。RT システムエディタは前者の静的システム構成を支援するツールである。本研究項目の最終目標は、RT システムエディタの修正・更新・機能拡張をすすめ、信頼性の高いツールを本プロジェクト外部に対して公開するとともに事業化を行うことである。

RT システムエディタ（RTSystemEditor）概要

RT System Editor は、OpenRTM-aist に含まれる開発ツールの 1 つであり、ネットワーク上で動作中の RTC をグラフィカル操作する機能を持つ。Eclipse 統合開発環境のプラグインとして作成されており、Eclipse 上にて既存のプラグインとシームレスに操作を行うことが可能である。プロジェクト開始前から同等の機能を持つ RtcLink と呼ばれるツールがあったが、OpenRTP ツールチェーンに組込むために、いくつかの点で修正を行った上で、名称も RTSystemEditor と変更した。

RTsystemEditor は図 62 のような操作画面を持つ。基本的な操作は、左側のネームサービスビューに表示されている実行中の RTC を中央のエディタにドラッグアンドドロップし、RTC のアクティブ化・非アクティブ化、ポートの接続、コンフィギュレーション・パラメータの操作等を行う。

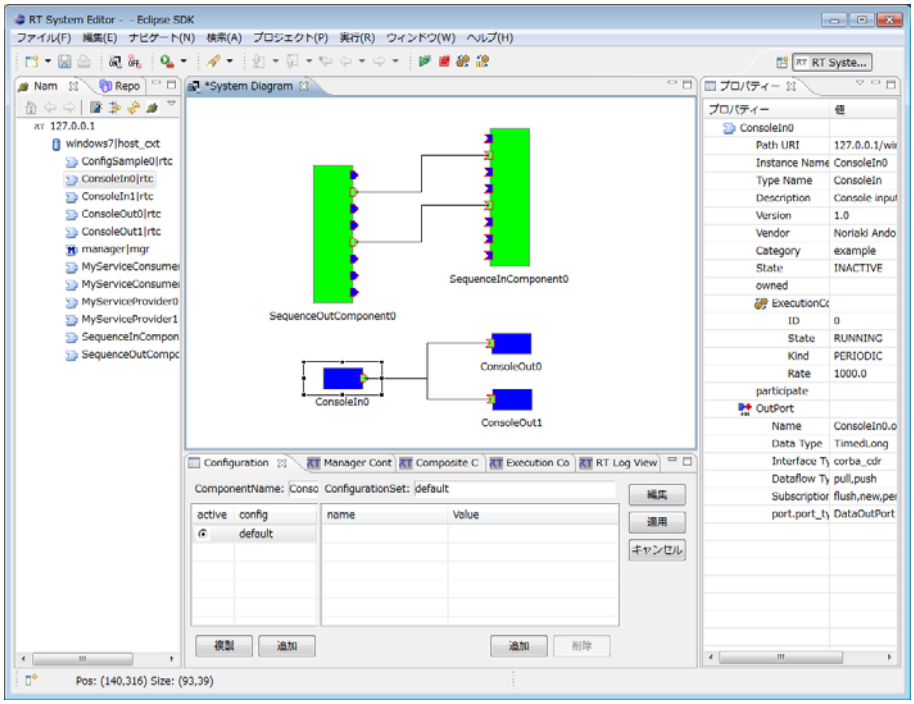

図 62 RTsystemEditor の操作画面

表 14 に RTSystemEditor の機能一覧を示す。

表 14 RTSystemEditor の機能

№	機能名称	機能概要
1	コンポーネントコンフィグレーション表示/編集機能	選択したコンポーネントのコンフィギュレーションプロファイル情報をコンフィグレーションビューに表示し編集する。
2	コンポーネント動作変更機能	選択したコンポーネントの動作を変更する。
3	RT システム組み立て機能	システムエディタ上でシステムの組み立てを行う。
4	システムセーブ/オープン機能	システムエディタの内容を RTS プロファイルとしてセーブする。RTS プロファイルをシステムエディタでオープンする。システムのポート接続、コンフィグレーションを変更しない）
5	システム復元機能	RTS プロファイルをシステムエディタでオープンし、プロファイルの内容を元にシステムを復元する。（プロファイルの内容でシステムのポート接続、コンフィグレーションを再構築する）

RT System Editor は、この OpenRTM-aist に含まれる開発ツールの１つであり、RTC をリアルタイムにグラフィカル操作する機能を持っています。また、その名前のとおり Eclipse 統合開発環境のプラグインとして作成されており、Eclipse 上にて既存のプラグインとシームレスに操作を行うことができます。

RTSystemEditor の各種ビュー概要

Eclipse の画面は種々の機能を提供するビューやエディタと呼ばれるサブウインドウ（ペインとも呼ぶ）から構成される。表 14 に RTSystmEditor のビュー一覧、図 63、図 64、図 65 にそれぞれのビューの外観を示す。

表 15 RTSystemEditor のビュー

№	ビュー名	説明
1	ネームサービスビュー	RTC が登録されているネームサービスの内容をツリー表示します。
2	コンフィグレーションビュー	選択されている RTC のコンフィグレーション情報を表示/編集します。
3	マネージャコントロールビュー	選択されているマネージャを制御します。
4	複合コンポーネントビュー	選択されている複合 RTC のポート公開情報を表示/設定します。
5	プロパティビュー	選択されている RTC のプロファイル情報を表示します。
6	システムエディタ	RTC をグラフィカルに表示し、RT システムを作成します。
7	オフラインシステムエディタ	RT リポジトリやローカルの RT コンポーネント仕様ファイルの内容をグラフィカルに表示し、RT システムを作成します。

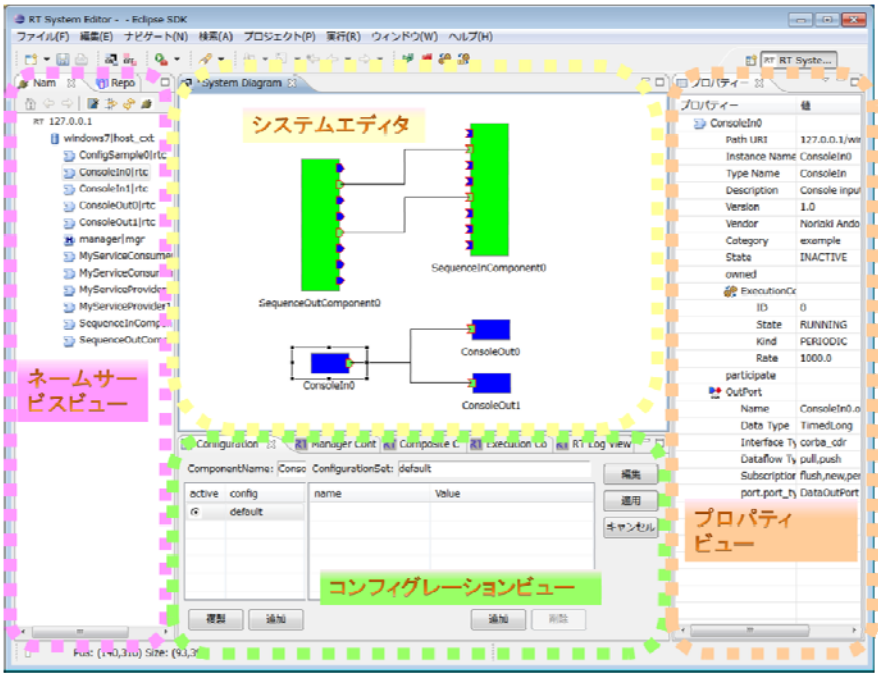

図 63 ネームサービス・コンフィギュレーション・プロパティ各ビューおよびシステムエディタ

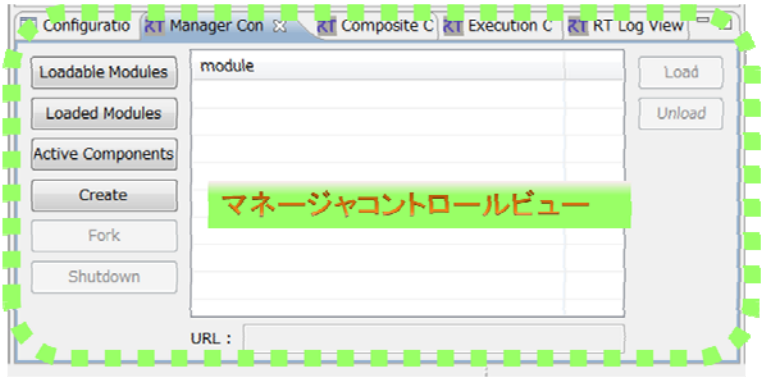

図 64 マネージャコントロールビュー

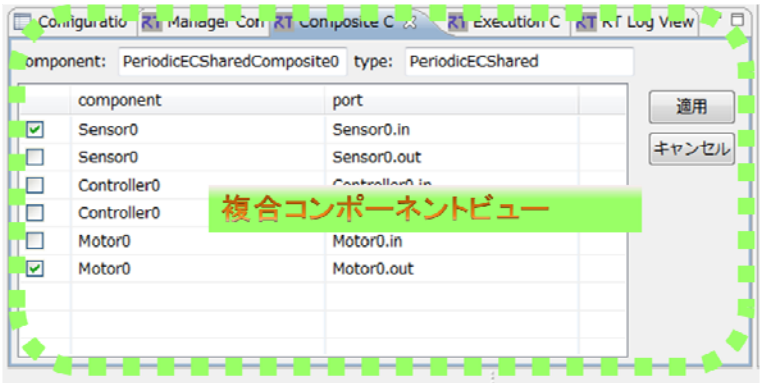

図 65 複合コンポーネントビュー

以下、操作の基本となるネームサービスビュー、システムエディタ、コンフィギュレーションビューについて説明する。

ネームサービスビュー

RTC をネットワーク上で動作させる際には、そのオブジェクトリファレンスと名前を一括で管理するネームサービスが必要となる。すべての RTC は起動後ネットワーク上に存在するネームサービスに名前とオブジェクト参照を登録する。アプリケーションや RTSystemEditor は登録されたオブジェクト参照を元に、各種コマンドを RTC に送ることで様々な操作を行う。

予めネットワーク上に 1 つ以上のネームサーバを起動しておき、起動する RTC の設定ファイル（通常は rtc.conf）に当該ネームサーバのアドレス（およびポート番号）を記述しておく。こうすることで、起動する RTC の名前とオブジェクト参照が自動的にネームサーバに登録される。

ネームサービスビューでは、ネームサーバに登録されている RTC の一覧を見ることができる。ネームサービスビューの接続アイコン（図 66）を押すと接続ダイアログ（図 67）が表示されるので、上記で設定したネームサーバのアドレスをダイアログに入力する。

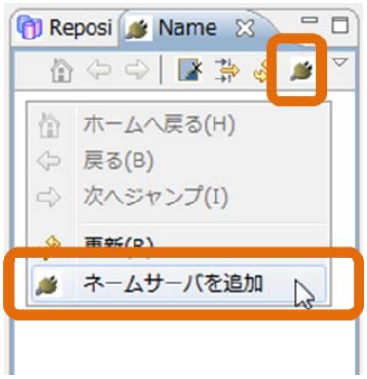

図 66 ネームサービスへの接続

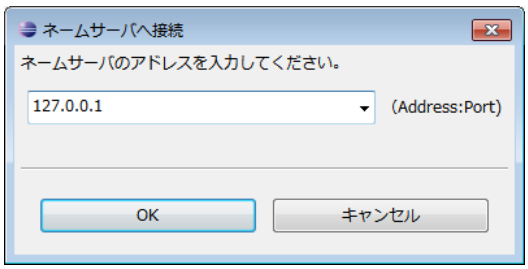

図 67 ネームサーバアドレスの入力

ネームサーバが正常に動作しており、かつ RTC が登録されている場合、図のように RTC の一覧がビューに表示される。

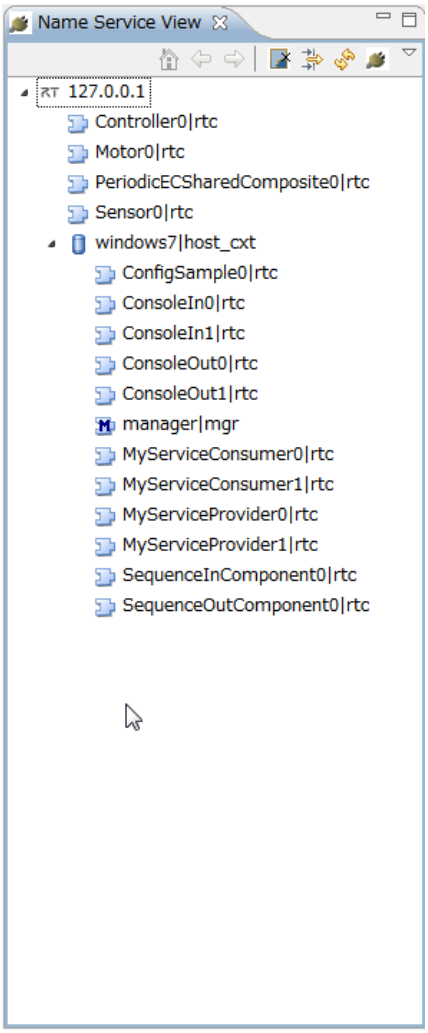

図 68 ネームサービスビューに登録された RTC 一覧

ツリー表示されるオブジェクトの種類とそのアイコンを表 16 に示す。

表 16 ネームサービスビュー上のアイコンとオブジェクトの種類

№	アイコン	種類（KIND）	名前
1		host_cxt	ホストコンテキスト
2		mgr_cxt	マネージャコンテキスト
3		cate_cxt	カテゴリコンテキスト

4		mod_cxt	モジュールコンテキスト
5		上記以外	フォルダ（上記以外のコンテキスト）
6		なし	RTC
7		なし	マネージャ
8		なし	オブジェクト（RTC 以外のオブジェクト）
9		なし	ネームサーバにエントリされてはいるが、実体のオブジェクトにアクセスできないゾンビオブジェクト

加えてネームサービスビューでは以下の操作を行うことができる。

- 複数のネームサーバの表示
- ネームツリーの階層ごとの表示
- 表示のフィルタリング
- ネームサーバエントリの削除
- オブジェクトエントリの追加・削除
- コンテキストの追加・削除
- ゾンビオブジェクトの削除
- RTC の基本操作（アクティブ化・非アクティブ化・リセット等）

詳細については、OpenRTM-aist オフィシャル Web サイト（http://www.openrtm.org）の RTSystemEditor のドキュメントを参照されたい。

システムエディタ

システムエディタでは、RTC の状態（Inactive・Active・Error）がアイコンの色（青・緑・赤）でリアルタイムに表示される。またポート間の接続、コンフィギュレーションの変更、RTC の状態を変更することでシステム構築、動作検証を行う。

システムを構築するためには、中央のシステムエディタに必要なコンポーネントを配置する必要がある。RTC をシステムエディタに配置するには、ネームサービスビューから RTC をドラッグ＆ドロップする（図 69）。

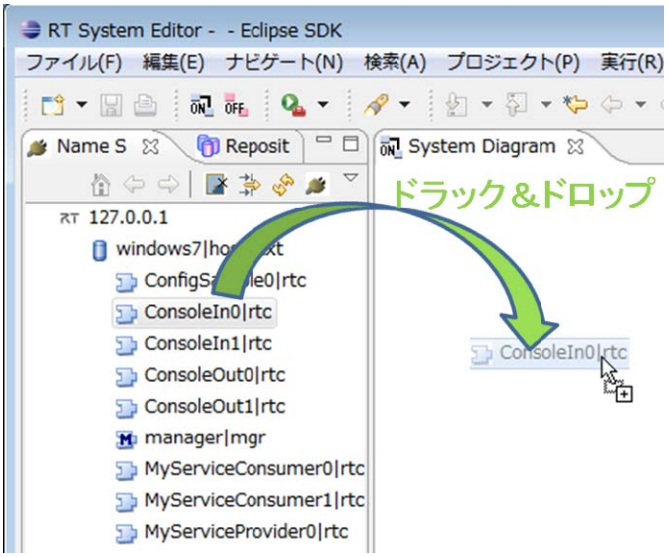

図 69 ドラッグ＆ドロップによる RTC の配置

配置した RTC 同士は、必要に応じてポート間を接続する。ポートの接続は、図 70 のようにポートとポートをドラッグ＆ドロップすることにより行われる。

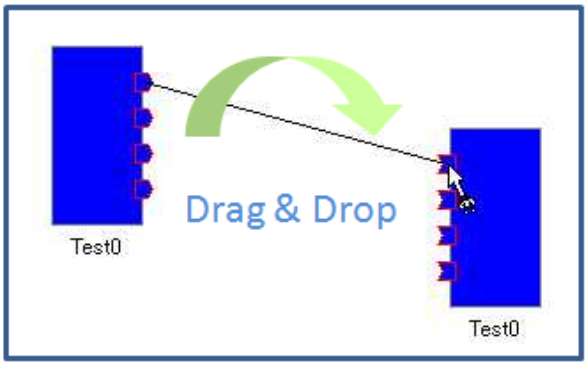

図 70 ドラッグ＆ドロップによるポート接続

ドラッグ＆ドロップ終了後、接続に必要な情報の入力を促すダイアログが表示される（図 71）。ここで設定される情報は ConnectorProfile と呼ばれ、送受信方向、データ転送のタイミング、バッファリングの設定など様々な設定を行うことができる。ConnectorProfile は、それぞれのポートが必要とする条件を満たすように作成される必要があるが、このダイアログは必要な条件を満たす値のみが入力されるようプルダウンで促す。

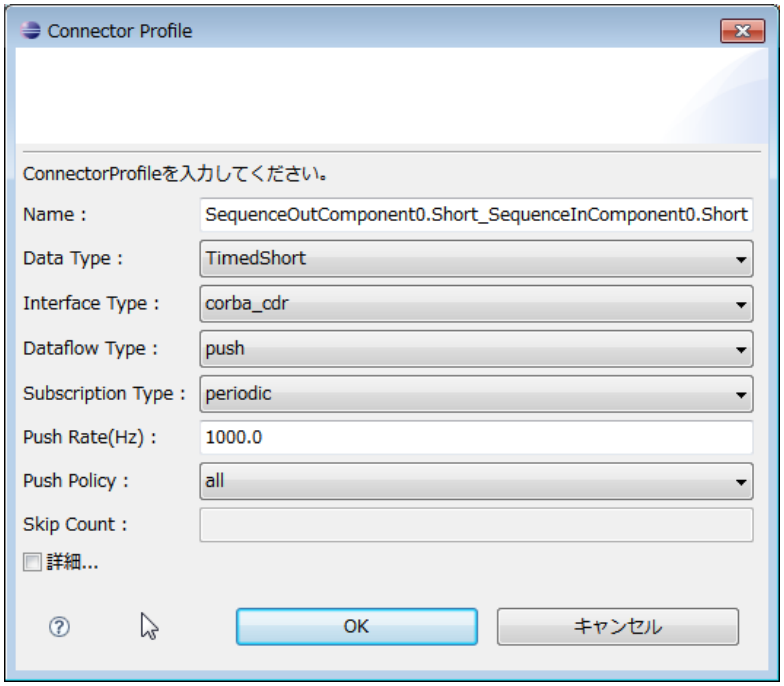

図 71 ConnectorProfile の作成ダイアログ

RTC のポートを接続してシステムが構成できたら、次にコンポーネントをアクティブ化する。RTC をアクティブ化する方法には幾つかあり、図 72 に示すように、アクティブ化（あるいは非アクティブ化）したい RTC を右クリックし、出てくるコンテキストメニューから「Activate」を選択することで RTC をアクティブ状態することができる。コンテキストメニューにはこの他「Deactivate」、「Reset」、「Finalize」、「Exit」、「Start」、「Stop」メニューがあり、それぞれ RTC に対してコマンドを送ることができる。

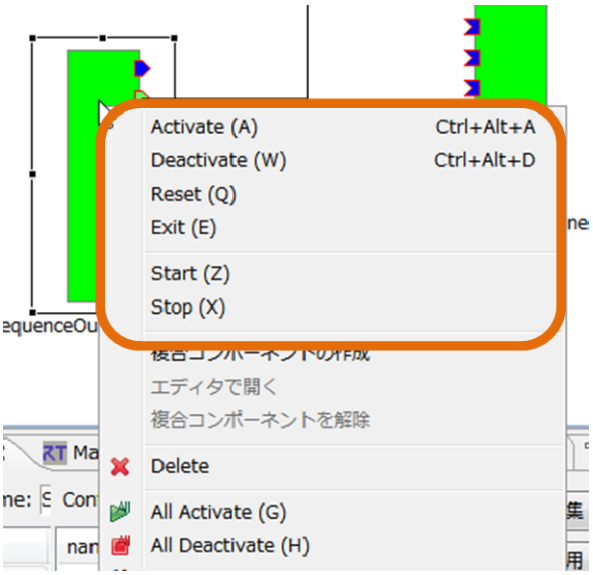

図 72 コンポーネントの状態の変更

この他、システムエディタの RTC 以外の部分を右クリックすると現れるコンテキストメニューを使うと、図 73 のように現在表示しているシステムエディタ上の RTC すべてを Activate（All Activate）また Deactivate（All Deactivate）することもできる。同時に表示される All Start および All Stop は各コンポーネントのデフォルトコンテキストのスレッドを開始・停止するためのコマンドであり、通常は使用しない。これら、All Activate/Deactivate および All Start/Stop メニューは、画面上部のメニューバーにも同一のものが常に表示されている。

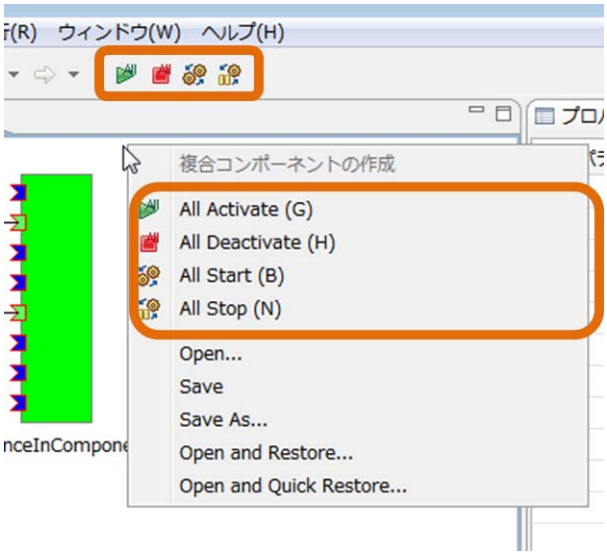

図 73 全 RTC の Activate/Deactivate を行うメニュー

コンフィギュレーションビュー

RTC におけるコンフィギュレーション（またはコンフィギュレーション・パラメータ）とは、RTC 内部の特定のパラメータを外部から参照・変更できるようにしたものである。これを操作するためのビューをコンフィギュレーションビューと呼ぶ。コンフィグレーションビューでは、RTC を選択すると図 74 のように RTC のコンフィギュレーションセット（図左）およびコンフィギュレーション・パラメータ（図右）が表示される。

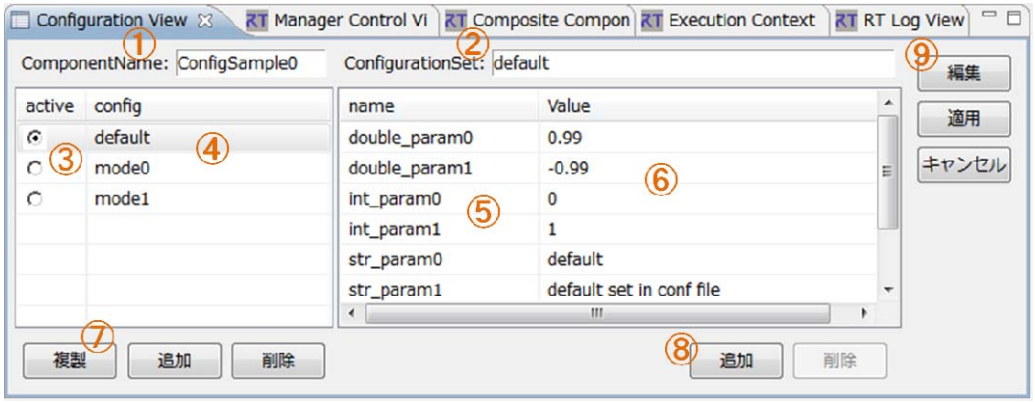

図 74 コンフィギュレーションビュー

変更したパラメータを⑥で選択し、パラメータの値を変更、⑨の更新ボタンを押すことによりパラメータの変更を行う。または⑨の編集ボタンを押し、図のようなダイアログを表示させ、パラメータの変更を行う。

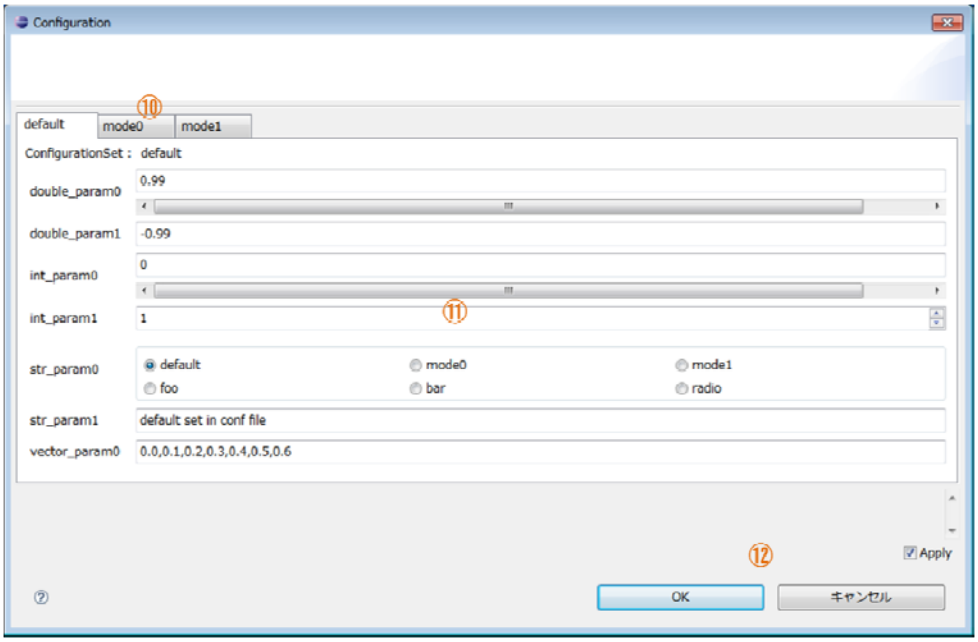

図 75 コンフィギュレーション・パラメータ編集ダイアログ

RTC 実装時に、コンフィギュレーション・パラメータに適切なサブオプションが指定されていれば、パラメータを「スライダ」「スピンボタン」「ラジオボタン」などで変更することができる。その際、ダイアログ右下の「Apply」チェックが入っている場合、操作ごとに連続して自動的にパラメータの更新が行われる。すなわち、スライダなどで値を変更すると、その RTC 内の値がリアルタイムに更新される。また、各パラメータは事前に制約条件を与えることもでき、不適切な値が入力された場合は、図 76 のようなダイアログが出てユーザに注意を促す。

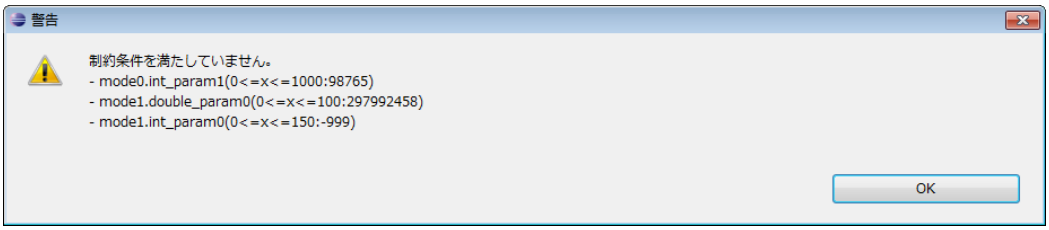

図 76 制約条件を満たさない値が入力された場合の警告ダイアログ

追加機能（Observer）

RTSystemEditor は管理しているすべてのコンポーネントについて、その状態（ライフサイクル状態、接続情報、コンフィギュレーション・パラメータ等）を把握するために、一定周期でポーリングにより情報取得を行なっている。状態の変化が発生しなくとも常に情報取得を行っているため、管理するコンポーネントの数が増えた場合に処理時間の増加につながる。また、状態の変化を周期ごとにしか把握する

ことができないため、表示と実際の状態の間に不整合が発生する場合がある。これを解決するために、バージョン 1.1 において、OpenrRTM-aist と RTSystemEditor に ComponentObserver と呼ばれる仕組みを導入した。
RTC はサービスポート以外にポートに付属しないサービスインターフェース：SDOService を持つことができる。状態を通知するためのオブザーバーインターフェースを定義し、RTC 側に ComponentObserver 要求(Required)インタフェース、RTSystemEditor 側に ComponentObserver 提供（Provided）インタフェースを持たせる。定義した ComponentObserver インタフェースの IDL を図 77 に示す
RTSystemEditor は管理する RTC に対して、SDOService として ComponentObserver をアタッチ可能か問い合わせる。アタッチできない RTC（バージョン 1.1 未満)については従来通りポーリングで対処する。アタッチできる RTC に対しては、ComponentObserver オブジェクトを作成、ターゲットの RTC にアタッチする。これにより、RTC は内部状態の変更が発生した時にのみ RTSystemEditor に状態の変化が発生したことと、どのような種類の変更が発生したかを通知することができる。ポーリングに比べて、状態変更発生時に直ちに変更が通知されるためタイムラグが少なく、変更が発生した時のみ通信が発生するため効率的である。図 78 に ComponentObserver のシーケンス図を示す。

```
// 状態の種類
enum StatusKind
{
   COMPONENT_PROFILE, // RTC の Profile が変化した
   RTC_STATUS, // RTC の LifeCycle 状態が変化した
   EC_STATUS, // EC に関する変更があった
   PORT_PROFILE, // Port に関する変更があった
   CONFIGURATION, // Configuration に関する変更があった
   HEARTBEAT, // Hearbeat 信号
  STATUS_KIND_NUM
};
// ComponentObserver インタフェース
interface ComponentObserver
    : SDOPackage::SDOService
{
   oneway void update_status(in StatusKind status_kind, in string hint);
};
```

図 77 ComponentObserver インタフェース

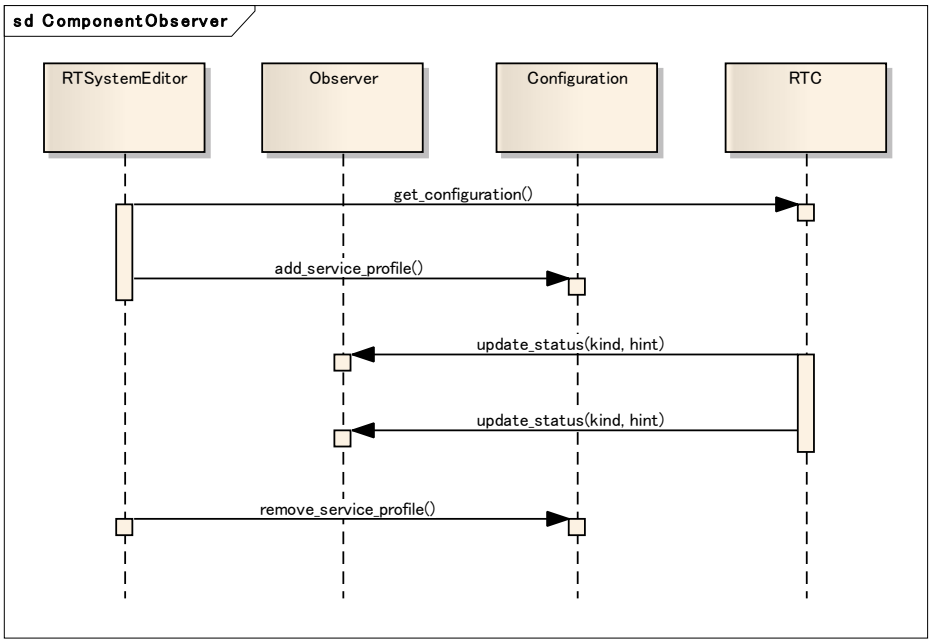

図 78 ComponentObserver の動作シーケンス

リリース履歴

- 平成 20 年 9 月 8 日：RTCBUilder・RTSystemEditor version 0.4 リリース
- 平成 22 年 2 月 9 日：RTCBUilder・RTSystemEditor version 1.0-RC1 リリース
- 平成 22 年 6 月 1 日：RTCBUilder・RTSystemEditor version 1.0-RELEASE リリース
- 平成 23 年 5 月 2 日：RTCBUilder・RTSystemEditor version 1.1-RC1 リリース

RTSystemEditor のまとめ

以上、RT システムを構築や RTC の操作を行う RTSystemEditor の機能について説明した。本プロジェクトで作成したこのツールは、14000 回以上ダウンロードされ、プロジェクト参加組織のみならず一般ユーザにも広く使われ、RT システム作成のための標準ツールとして定着したといえる。本ツールは、オープンソースライセンスと個別ライセンスのデュアルライセンスで公開を行っている。ツールのサポートに関して、開発者、ユーザのコミュニティを形成し継続的に実施している。また、個別ライセンスにより、商用化も視野にいれた開発を実施した。
また、プロジェクト期間中にユーザからの要求に応じて、後述するコマンドラインベースのツール rtshell の開発も実施した。以下に、rtshell について述べる。

rtshell 概要

RTSystemEditor を利用すれば、GUI からネットワーク上の RTC を操作し、システムを構築することができる。また、システム情報のセーブ・リストア機能を用いれば、システムの起動を自動的に行うこともできる。しかしながら、Eclipse は使用メモリ量が多く、高速の CPU を必要とするアプリケーションであり、低速の PC から RTC を操作したい場合には向かない。また、GUI ツールである RTSystemEditor に対して、CUI（Command-line User Interface）で対話的に RTC を操作したいという要望は依然として強い。また、システムの運用を自動化したい場合、一般には GUI ツールは適しておらず、スクリプトによる RTC の操作が求められる。
こうした背景から、RTC および RT システムの操作を行うスクリプト用ライブラリ rtctree/rtsprofile Python モジュールおよび、コマンドセット rtshell を開発した。rtctree/rtsprofile を用いると、Python スクリプトから RTC オブジェクトの取得、RTC の各種操作、システム構成情報の取得やリストア等を容易に行うことができる。また、rtshell コマンド群は、Linux や Windows のコマンドラインシェルから利用でき、対話的に RTC の状態の取得や操作、システム情報の保存・リストアを行うことができる。rtshell コマンド群は比較的リソースの乏しい PC でも動作し、開発から運用時までさまざな場面で利用できるツールである。

rtshell コマンド群

rtshell は表 17 に示す多くのコマンド群から構成されている。
RTC の操作に関するコマンドは、予め設定されたネームサーバに登録された RTC を参照し、ターゲットとする RTC をコマンドの引数として与え各種操作を行う。ポートの接続等のコマンドは、RTC に加えてポート名も引数に指定する。システム全体に対する操作コマンドは、システムのプロファイルである RTS Profile ファイルを与え、そこに指定されている RTC に対して各種操作を行う。
UNIX 系のシステムにおいては、各コマンドに man によるマニュアルページが用意されているので、詳細はマニュアルを参照されたい。

表 17 rtshell コマンド一覧

コマンド名	機能
rtact	RTC を Activate する。Inactive 状態の RT コンポーネントを activate する。
rtcat	RTC の情報を表示する。 RT コンポーネント、ポート及びマネージャのメタデータを読んでターミナルに表示する。
rtcheck	RT システムをチェックする。実行中の RT システムと RTSProfile とで矛盾がないかをチェックする。正しくない状態にあるコンポーネ

	ントや間違った接続などのようなエラーを報告する。
rtcomp	コンポジットコンポーネントの管理。実行している複数のコンポーネントを一つのコンポジットコンポーネントとして構成し、選択されたポートを外部に公開する。既存のコンポジットコンポーネントに新しいメンバーを追加、またはメンバーを削除する。
rtcon	二つ以上のデータポートやサービスポートを接続する。
rtconf	コンフィグレーションパラメータの管理。コンフィグレーションパラメータとセットを表示、編集する。
rtcryo	実行中の RT システムを RTSProfile に保存する。コンポーネント間の接続とコンポーネントの現在のコンフィグレーションパラメータを保存する。接続されていないコンポーネントは保存されない。
rtcwd	RTC ツリーの中の現在のワーキングディレクトリを変更する。
rtdeact	Active 状態の RT コンポーネントを deactivate する。
rtdel	ネームサーバから登録済みオブジェクトを消す。このコマンドでオブジェクト自身は終了されない。
rtdis	ポートの間の接続を削除する。一つのポートから全ての接続を消すこともコンポーネントから全ての接続を消すことも可能。
rtdoc	RT コンポーネントのドキュメンテーションを表示する。 ある RT コンポーネントは内部的にドキュメントを含んでおり、コンフィグレーションセットに保存されている。このコマンドでそのドキュメンテーションを複数のフォーマットで表示することができる。
rtexit	実行中の RT コンポーネントを終了させる。コンポーネントは終了手続きをして終了する。
rtfind	RTC ツリーのネームサーバ上でコンポーネントやマネージャなどを探索する。
rtinject	値を一つ以上のポートに送信する。デフォルトは一回のみ送信するが、複数回や定期的に送ることも可能。目的のポートに対して接続を作成する。
rtlog	コンポーネントがデータポートで送るデータをログファイルに保存したり、再生したりする。
rtls	RTC ツリーのディレクトリにあるオブジェクトをリストする。デフォルトは現在のワーキングディレクトリをリストする。
rtmgr	マネージャを介してコンポーネントと共有モジュールの管理を行う。マネージャにロードされた共有モジュールからコンポーネントをインスタンス化する。
rtprint	複数のアウトポートが送るデータを標準出力に表示する。
rtpwd	現在のディレクトリを表示する。
rtreset	エラーステートにある RT コンポーネントをリセットする。
rtresurrect	RTSProfile ファイルをロードして実行中のコンポーネントを使って RT システムを復元する。コンポーネントの間の接続とコンポーネントのコンフィグレーションパラメータは RTSProfile のものが反映される。

Rtstart	すべてのコンポーネントを activate することによって RTSProfile に保存された RT システムを起動する。コンポーネントは RTSProfile が指定された順番に activate される。
rtstodot	Graphviz の dot フォーマットで RT システムをグラフとして表示する。RTSProfile が指定されていない場合、stdin から読み込む。
rtstop	すべてのコンポーネントを deactivate すること似よって RTSProfile に保存された RT システムを停止する。コンポーネントは RTSProfile が指定された順番に deactivate される。
rtteardown	RTSProfile に保存された RT システムのすべての接続を削除する。
rtvlog	個別の RT コンポーネントのログを取得して表示する。RT コンポーネントには SDO の Logger インタフェースに対応している必要がある。ログはターミナルに出力される。

YouTube によるチュートリアル公開

rtshell の使い方をわかりやすく説明しユーザを広げるため、YouTube 上に OpenRTM-aist のチャンネルを作成し、そこで rtshell のチュートリアルビデオを公開した。英語版では 887 回、日本語版では 534 回（平成 24 年 4 月現在）の閲覧があった。また、RT ミドルウェア講習会においても rtshell を取り入れ、利用法を解説し実習中にユーザに利用してもらうことで、普及を図ってきた。

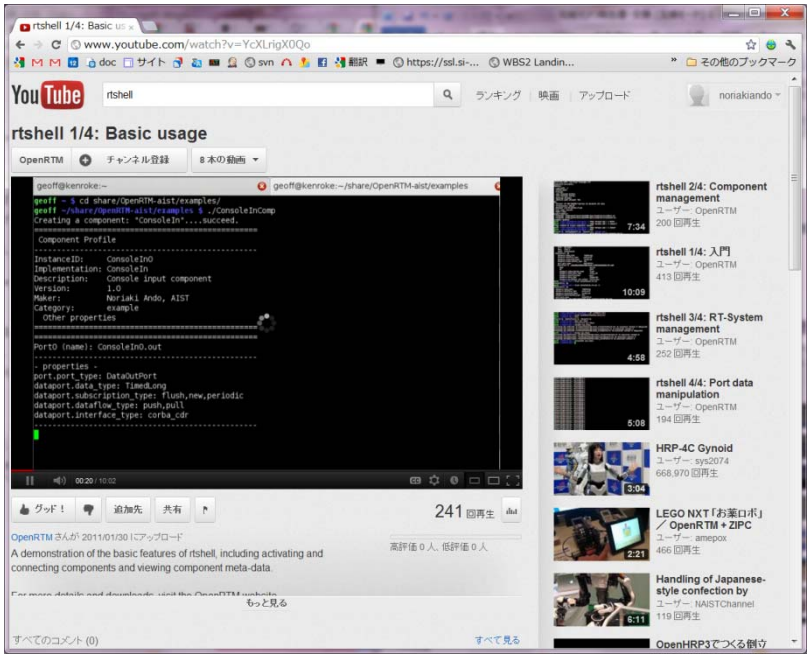

図 79　YouTube 上の rtshell チュートリアル

リリース履歴

- 平成 22 年 1 月 27 日：（rtshell の前身）rtcshell/rtsshell-1.0 リリース
- 平成 22 年 1 月 27 日：rtctree/rtsprofile-1.0 リリース

- 平成 22 年 6 月 1 日：（rtshell の前身）rtcshell/rtsshell-2.0 リリース
- 平成 22 年 6 月 1 日：rtctree/rtsprofile-2.0 リリース
- 平成 23 年 2 月 7 日：rtshell-3.0、rtctree/rtsprofile-3.0 リリース

rtshell のまとめ

rtshell はコマンドラインベースで RTC の種々の操作ができるツールとして、とくに Linux ベースでシステムを開発するユーザに好んで利用されている。また、rtctree/rtsprofile により、システムや RTC の複雑な操作をスクリプトとして記述し、システムの起動や運用の自動化を容易に実現できるようになった。これにより、GUI による RTC 開発・システム開発から、スクリプト・アプリケーションプログラムにより自動実行・運用に至るまでの RT システム開発・運用フェーズすべてをサポートするプラットフォームを実現した。本研究開発項目は、当初の実施計画にはなかったがプロジェクト期間中にユーザからの要求に基づき実施したものである。RTSystemEditor とともに、多くのユーザから利用されるツールを実現することができた。

（c）RT ミドルウェアの開発

（c-1）RT ミドルウェアの各種 OS／言語対応

本項目では、RT ミドルウェアを利用できるプラットフォームを拡充するため、RT ミドルウェアの各種 OS／言語対応を行う。

OS 対応としては、リアルタイム OS である VxWorks 上で動作する RT ミドルウェアを研究開発する。言語対応としては、Microsoft 社の .NET（ドットネット）環境で動作する RT ミドルウェア（OpenRTM.NET）の研究開発を行う。

その他の OS／言語対応については、ソフトウェア業界の動向、本プロジェクトの各研究項目実施機関からの要望を踏まえ、必要に応じて対応するプラットフォームを追加する。最終目標は、VxWorks および.NET 環境対応版の RT ミドルウェア以外に１つ以上の各種 OS／言語対応のＲＴミドルウェアのラインナップを追加する。本研究項目の最終目標は、各種 RT ミドルウェアに対して、機能向上、バグフィックスを行い、本プロジェクト外部に対して公開するとともに、事業化を行うことである。

VxWorks 版 RT ミドルウェア

これまでに開発された RT ミドルウェアは、Linux や Windows などの汎用 PC 上で動作するものであったが、今後はリアルタイム OS、組込みマイコンなどの様々なアーキテクチャへの適用が期待されている。そこで我々は、リアルタイム OS である VxWorks 上に産総研が開発している、

OpenRTM-aist-1.0.0-RELEASE 版を移植した。
VxWorks 版 RT ミドルウェア（OpenRTM-aist for VxWorks）のシステム構成を示す。

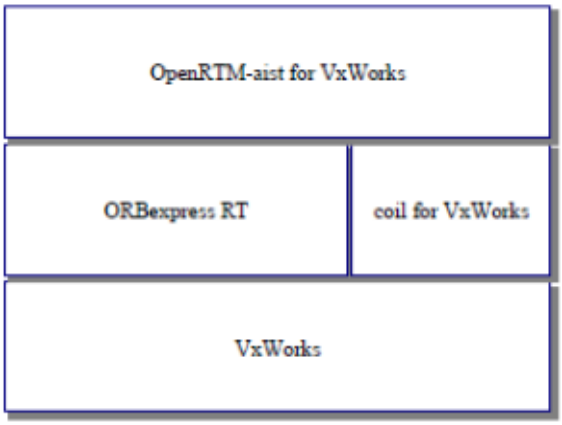

図 80　VxWorks 版 RT ミドルウェアのシステム構成

VxWorks は、WindRiver 社の開発するリアルタイム OS であり、産業用ロボットや航空宇宙機器など高い信頼性が要求される分野において広く利用されている。また、知能化プロジェクト内の開発対象となっている産業用ロボットにも利用されている。対応する VxWorks のバージョンは、メモリプロテクション機能を持ち、品質の高いアプリケーション開発が可能な 6.4 / 6.7 と、メモリプロテクション機能は持たないがパフォーマンス面で有利な 5.5.1 を対象とした。
ORBexpress RT は、OIS 社の開発する RT CORBA 1.0 Specification に準拠した CORBA 実装であり、航空宇宙や、原子力関連システムなどミッションクリティカルな分野で幅広く利用されている。オーバーヘッドが小さく、単純なソケット通信(TCP)と比較しても遜色がない。また、ライブラリのサイズは 200～300KB 程度と小さい。
OpenRTM-aist では、移植性を向上させるため、OS 抽象化層 coil (Common Operating-system Infrastructure Layer)を用意している。これは、スレッドやタイマーなど、システムコールを抽象化した汎用的なライブラリである。今回、VxWorks5.5.1 用、6.4 用のそれぞれの coil 実装を用意した。
VxWorks 版の RT ミドルウェアは、次に示す PowerPC 搭載の CPU ボード上で検証を行い、リアルタイム動作が行えることが確認できた。

Board	EK7 – ESM™ Starter Kit with EM8 / MPC 8540
CPU	PowerPC MPC8540 / 800 MHz
Memory	512MB DDR SDRAM
I/O	Gigabit / Fast Ethernet UART PCI SLOT VGA USB 2.0 IDE PC 104
Size	170mm × 150mm
Weight	465g

図 81　VxWorks 版 RT ミドルウェアの評価環境（その 1）

Board	WIND RIVER SBC8540
CPU	PowerPC MPC8540 / 800 MHz
Memory	512MB DDR SDRAM 64MB Flash DIMM 8MB Onboard Flash 64KB EEPROM
I/O	Gigabit / Fast Ethernet PCI SLOT RapidIO 8bit parallel bus 16pin JTAG COP Interface
Size	312mm × 128mm

図 82　VxWorks 版 RT ミドルウェアの評価環境（その 2）

.NET 対応 RT ミドルウェア（OpenRTM.NET）

我々は、OMG (Open Management Group)で標準化され、平成 20 年 4 月に仕様が公開された RTC Specification 1.04)準拠の RT ミドルウェアとして、OpenRTM.NET 1.0 を開発した。OpenRTM.NET 1.0 は Microsoft .NET Framework 上で動作する RT ミドルウェア実装であり、CORBA (Common Object Request Broker Architecture)だけでなく、SOAP によるインターネット経由の通信や、名前付きパイプによる高速な通信など、複数の通信方式を選択可能として

いる。また、アノテーションを活用することで、RT コンポーネントを簡略化した記述で実装することが可能となっている。OpenRTM.NET は次のような特徴を持つ。

- C#や Visual Basic を利用して、効率的に RT コンポーネントを開発することができる。
- インストーラがサポートされており、簡単に使い始めることができる。
- CORBA や WCF など、複数の通信ミドルウェアに対応している。

ここで、OpenRTM-aist は、通信ミドルウェアに CORBA (Common Object Request Broker Architecture)を利用しているため、ファイアウォールを通過した通信や、他のロボット用ミドルウェアへの透過的なアクセスを実現することは難しい。OpenRTM.NET では、RT コンポーネントフレームワークと通信ミドルウェアの構造を分離することにより、CORBA 以外の様々な通信ミドルウェアに対応できるようにしていることも大きな特徴である。
OpenRTM.NET は、CORBA PSM の実装に加え、 Microsoft の通信基盤技術である WCF(Windows Communication Foundation)向けの PSM を規定し実装を行っている。WCF では、SOAP や REST(REpresentational State Transfer)、名前付きパイプなど、複数の通信プロトコルを透過的に利用することが可能であり、WS-Security を利用したセキュアな通信も実現することができる。また、通信ミドルウェア層を抽象化していることにより、CORBA や WCF 以外の新たな通信ミドルウェアを利用するための拡張が可能となっている。

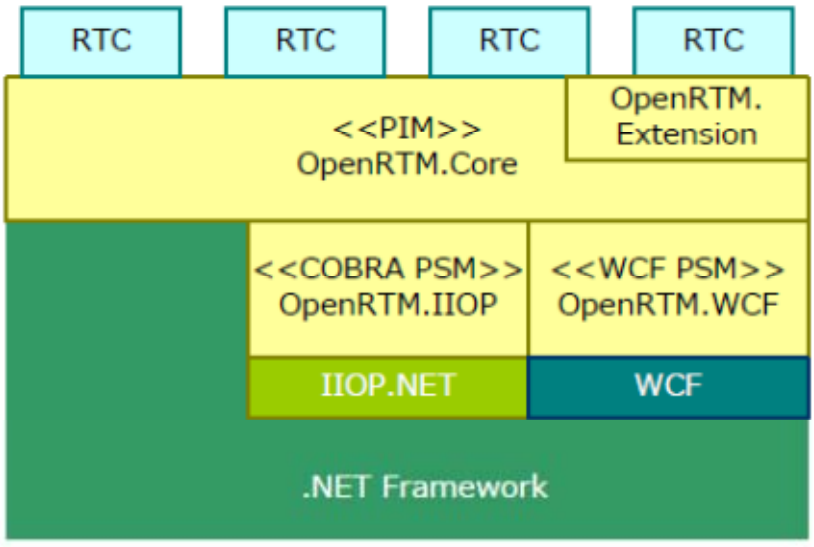

図 83　OpenRTM.NET のアーキテクチャ

通信ミドルウェアの抽象化を行った場合、通信ミドルウェアの実装に依存しない

形で、データやサービスのインタフェースを規定するための表現方式が必要となる。例えば、CORBA では IDL(Interface Definition Language)、SOAP では WSDL(Web Services Description Language)が利用されている。しかし、こういった IDL や WSDL のような形式を利用した場合、新たな記述方式を学習するためのコストが必要となる。また、開発の際に IDL コンパイル等の手間や、IDL とプログラミング言語間の実装の不一致が問題となる。そこで、OpenRTM.NET では、C#や Visual Basic など、RT コンポーネントを実装するプログラミング言語と同じ言語で共通インタフェースを記述する。ただし、共通インタフェースであることを表すための、カスタム属性(アノテーション)を付与することとしている。例えばデータ型の定義は、DataContract と DataMember を利用して、サービス型の定義は、 ServiceContract と OperationContract を利用して、次のように記述することができる。

```
[DataContract]
public class MyData {
  [DataMember]
  public int value;
}
[ServiceContract]
public interface MyService {
  [OperationContract]
  string echo(string message);
}
```

図 84　OpenRTM.NET におけるデータ型とサービス型の定義

この共通インタフェースを利用した RT コンポーネントを作成し、設定ファイル等で通信ミドルウェアに CORBA を選択して実行すると、OpenRTM.NET は、共通インタフェースの構造を解析し、IDL とスタブとスケルトンを自動生成する。さらに、スタブを利用したプロキシと、スケルトンを実装したアダプタも自動で生成される。この仕組みにより、通信部分のソースプログラムが動的に生成され、RT コンポーネントの実装を変更することなく、異なる通信ミドルウェア上で RT コンポーネントを動作させることが可能となる。

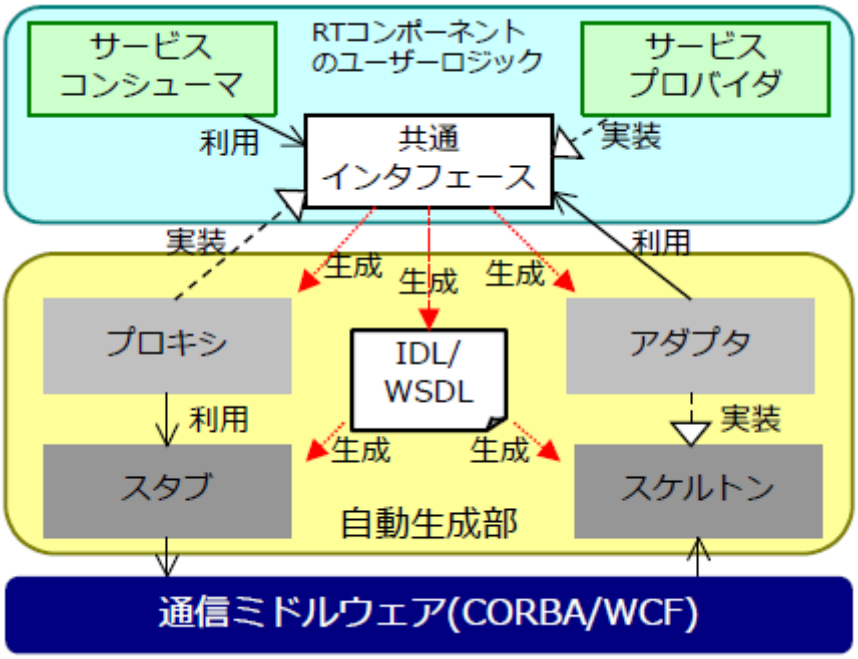

図 85　共通インタフェースによる通信部分のソースプログラムの自動生成

Android 版 RT ミドルウェア（RTM on Android）

異なる環境の知能モジュールをシームレスに接続可能とするため、様々な OS やプログラミング言語に対応した RT ミドルウェアを開発してきたが、急速に普及しているスマートフォンやタブレット PC への対応は、残念ながら遅れてきた。我々は、オープンソースで公開されている点、扱いやすいライセンス(Apache Software License v2)である点、豊富な Java API が公開されておりアプリケーションを開発しやすい点、などにおいて、今後益々の拡大が見込まれるプラットフォームである Android への適用は急務であると考え、Android 版の RT ミドルウェア（RTM on Android）を開発した。

Android のアーキテクチャを図に示す。

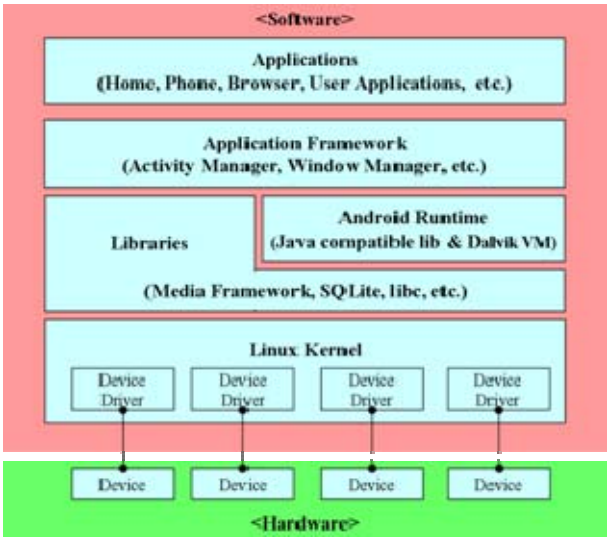

図 86　Android のアーキテクチャ

Android Runtime には、省メモリ、低スペック CPU などの環境に最適化された Dalvik VM が含まれている。各アプリケーションは、それぞれが独立した Linux のプロセスとなり、各アプリケーションに一つの Dalvik VM インスタンスが対応して動作する。一般的な Android アプリケーションは、UI を持つコンポーネントである Activity として実装される。Activity は画面の最上位に表示される場合にのみ動作することができ、他の Activity が上位に表示されたり、Back や Home キーを操作されたりした場合には休止状態となるライフサイクルが決められている。一方、UI を持たず、バックグラウンドで継続的に動作する Service も代表的なコンポーネント単位である。

ここで、RTC は、必ずしも UI を必要としない。また、他 RTC との通信は継続的に実施できる必要がある。よって、RTM on Android では、RTC が Android 上で Service として動作する実装を行った。そして、UI が必要な場合は別途 Activity を必要なだけ追加できるようにし、これら全体で一つのアプリケーション単位（apk）となるように考えられている。Service として RTC のコアな処理が動作することで、一つの Android 端末上にて複数の RTC を同時に Active な状態で稼動させることもできる。

RTM on Android は、OpenRTM の Android 実装を目指しており、CORBA 通信が必須である。標準の Android プラットフォームでは CORBA は未サポートであり、独自の実装が必要である。ただし、RTC を Service コンポーネントとしてバックグラウンドで継続的に動作させるため、実用化のためには、CORBA 実装を含め、システムリソースへの負荷を極力減らすことが重要である。この要求に応える CORBA 実装として、軽量 CORBA の RtORB を採用することとし、これを

Android にポーティングした。C 言語での実装を、Android 開発環境のツールキットとして Google 社より提供されている NDK(Native Development Kit)を利用することにより、ほぼそのままポーティングすることに成功した。ネーミングサービスとの I/F 実装についても、RtORB に同梱されている CosNaming.idl から C ソースを自動生成して同様にポーティングを行った。これらはともに、C のネイティブライブラリである.so 形式として生成し、組み込むことに成功した。

RTM on Android は、RTC 標準仕様 Ver1.0 に従った軽量 RTC の OpenRTM 拡張モデルとしての基本的な振る舞いに対応しつつ、実行時のシステムリソースへの負荷も軽い RTC が開発できることを目指している。よって我々は、OpenRTM の実装部分も IDL ファイルから C ソースを生成し、最低限のロジック実装を行い、やはり NDK の利用により.so 形式で組み込む実装を行った。

RtORB、CosNaming、 OpenRTM の実装をネイティブライブラリとして組み込めたことで、あとはこれらを利用する RTC を簡単に実装／実行できる仕組みがあれば良い。RTM on Android では、これらを利用して Service コンポーネントとして RTC が動作するための Java クラス群、およびネイティブライブラリで提供される OpenRTM などの実装を Java 層から利用するために JNI（Java Native Interfase）によりラッピングした独自ライブラリを実装することで、これを実現している。さらに、OpenRTM の基本データ型を扱うためのクラスや、RTC の状態遷移に伴い呼び出される Action の仕組み、またデータポートの入出力を行う仕組みなどを Java 層に実装した。この際、アプリケーション開発者が直感的かつシンプルに利用できる API を用意することで、簡単な最小限の記述を行えば RTC の実装が行えるように配慮している。

また、RTC には個々に任意の数のデータポートと対応するデータ型、コンフィグレーションの種類／データ型／初期値など、固有のプロファイル情報の指定が必須である。RTM on Android では、これらを定義する雛形 Java ソースを実装している。指定できるプロファイル情報は概ね OpenRTM-aist と同等であるが、CPU 負荷の調整を想定し、実行コンテキストの周期設定も有効となるようにした。さらに、Action の実装やデータポートからの入力時に呼び出されるコールバック実装を記述するための雛形 Java ソースを実装している。

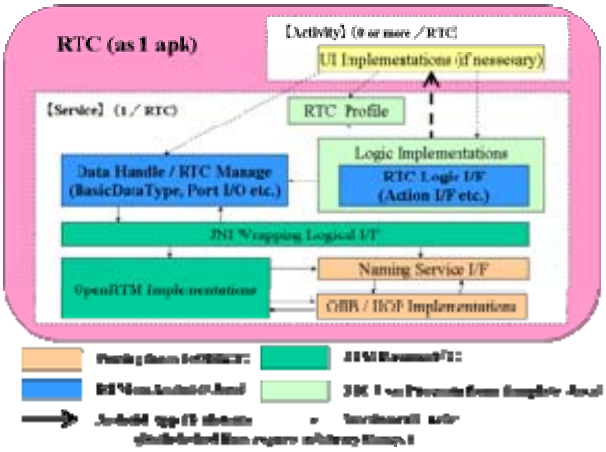

図 87　RTM on Android のアーキテクチャ

まとめ

本研究開発では、OS 対応としては、リアルタイム OS である VxWorks で動作する RT ミドルウェアおよび、スマートフォンやタブレットで採用されている Android で動作する RT ミドルウェアを開発した。言語対応としては、Microsoft 社の.NET 環境で動作する RT ミドルウェア（OpenRTM.NET）の開発を行った。産業用ロボットなどで数多く採用されている実時間 OS である VxWorks 向けの RT ミドルウェアを開発することは、RT ミドルウェアのビジネス利用を促進するために有効である。また、スマートフォンやタブレットなどの Android 端末は、ロボットと人とのユーザインタフェースとしてのニーズが見込まれるため、RT ミドルウェアが Android に対応していることは意義があると考えている。さらに、プログラミング言語として普及している.NET 環境に RT ミドルウェアを対応させることは、RT ミドルウェアの利用の敷居を下げ、RT ミドルウェアの普及に貢献した。

これらの RT ミドルウェアおよびマニュアルは、セックのロボットサイト（http://www.sec.co.jp/robot/index.html）にて無償公開している。NEDO 次世代ロボット知能化技術開発プロジェクトの参画機関を始め、その他の大学や企業でも利用されており、RT ミドルウェアの普及に貢献している。

（c-2）軽量 RT ミドルウェアの開発

ロボットに搭載される組込みプロセッサでは、使用できる主記憶や動作クロックが一般のパソコンよりも大きく制限されることが多い。このようなプロセッサ上は、現在実装されている RT ミドルウェアの性能を十分に発揮することができない。その要因として、コンポーネント間の通信で使用されているライブラリ等が汎用性に重点が置かれて実装されているために、組込み向けプロセッサのような制限

されたリソース上で十分なパフォーマンスが発揮できないということが考えられる。そこで、RT ミドルウェアをこのような組込み向けのプロセッサで軽快に動作させるために、以下の方法で RT ミドルウェアの軽量化を実施した。本研究項目の最終目標は、軽量 CORBA ンスマニュアル、移植マニュアルなどを整備し、軽量 CORBA ミドルウェア実装である OpenRTM-aist-1.0 および C 言語版 OpenRTM-aist-1.0 を実装し、外部に公開することである。

(A) 軽量 CORBA 実装による軽量化

RT ミドルウェア OpenRTM-aist では、知能モジュール間の通信に CORBA(The Common Request Broker Architecture)を用いている。CORBA は、国際標準機関 OMG(Object Management Group)で策定された分散オブジェクト技術の仕様であり、それに基づいたさまざまな実装が存在する。CORBA は、一般的な分散オブジェクト技術に関わる仕様が作成されており、RT コンポーネントの要求する機能以上のさまざまな機能が実装されている。そのため、既存の CORBA ライブラリを用いると、組み込み機器のような省資源の環境で動作させることは困難である。そのため、RT ミドルウェアが提供する機能のみを抽出した軽量 CORBA ライブラリ RtORB を実装し、それを通信部分として用いた RT ミドルウェアの軽量化を行った。以下のでは、RT ミドルウェアの軽量化のために実装した軽量 CORBA ライブラリ RtORB について述べる。

RtORB の基本設計

RT ミドルウェアでは、RT コンポーネント開発者に対して、CORBA に関するほとんどの API を隠蔽されている。そのため、RT ミドルウェアで提供する API レベルで同等の機能を実装することで十分である。また、RT コンポーネントの動作を考えると RT ミドルウェアで利用している CORBA の機能は、静的な分散オブジェクトモデルのみであり、さらに、オブジェクト間の通信には IPv4 の TCP/IP ソケットを利用した IIOP のみであった。そこで、RtORB では、以下の基本方針を設定する。また、RtORB の基本設計には、CORBA-3.0.3 の仕様を用いることにする。

基本方針

(A) 分散オブジェクトは静的なオブジェクトのみとし、オブジェクトごとのポリシ設定はしない
(B) ORB および POA のインタフェースは実装するが、実行時に生成可能な POA は RootPOA のみとする
(C) オブジェクト間の通信プロトコルは、IIOP-1.0 のみを使用し、同一プ

ロセス内での CORBA オブジェクト間の通信は IIOP を使用せず、関数呼び出しを直接実行する。

(D) 使用できるデータ型は、RT ミドルウェアのデータポートで利用しているデータ型のみとする

(E) 実行時の計算資源を節約するために、できる限りスレッドを使用せず割り込み処理を利用する。

以下のでは、上記の基本方針に関する詳細を述べる。

OMG で規定されている CORBA の仕様には、静的起動と動的起動の 2 つの分散オブジェクトが定義されている。どちらのオブジェクトも CORBA クライアントは、CORBA サーバの参照へのアクセスを取得し、要求を満たすオペレーションを呼びだし、その要求を実行するという基本的な枠組みは変わらない。静的起動の CORBA オブジェクトの場合には、CORBA クライアントは、IDL コンパイラによって事前に生成されたスタブ（Stub）を介して CORBA サーバへの要を処理し、呼び出す必要があるオペレーションはすべてアプリケーションのコンパイル時に認識している。一方、動的起動の CORBA オブジェクトの場合には、CORBA クライアントは、そのアプリケーションのコンパイル時には CORBA オブジェクトのインタフェースを認識している必要はなく、インタフェースリポジトリから CORBA サーバへ要求に必要なオペレーションインタフェースを取得し、CORBA サーバへの要求を動的に生成、実行を行なっている。そこで、RtORB では、基本方針(A)の基づき静的起動のみをサポートした CORBA オブジェクトのみを実装する。

また、CORBA では、CORBA サーバーアプリケーションを管理する機能として、BOA (Basic Object Adaptor)と POA (Portable Object Adaptor)の 2 つの種類が定義されている。BOA は CORBA-2.1 までの仕様で定義されていた基本的なサーバーアプリケーションのインタフェースであり、現在利用可能な CORBA ライブラリのほとんどがこの機能をサポートしている。POA は、CORBA-2.2 からの仕様で提起さてたサーバーアプリケーションのインタフェースであり、BOA と比較してサーバーオブジェクトごとに活性化た生存期間、スレッド処理などの様々なポリシを定義することができ、多様な管理を実行することができる。RT ミドルウェアの実装では、POA を用いて実装されているが RT ミドルウェアでは、コンポーネント単位で CORBA サーバを管理、実行しているため複数の POA を単一コンポーネントで使用するようになっておらず、BOA も利用されていない。そこで基本方針(B)に基づき、

RtORB では POA のみを実装し、単一のポリシのみを定義、実装を行う。

次に、CORBA オブジェクト間の通信に関して述べる。CORBA オブジェクト間の通信に関しては、仕様書では CORBA 以外に分散オブジェクトへのプロキシに関する様々な定義が記載されている。また、CORBA のオブジェクト間の通信に関しては、現在でも共有メモリやマルチキャストプロトコルへの対応など CORBA オブジェクトで利用される通信方法に関して拡張が続けられている。RT ミドルウェアでは、コンポーネント間はイーサネットで接続し動作させることを基本にしており、同期的な通信のみを取り扱っている。そこで、RtORB では、CORBA の通信方法として、コネクション型の TCP/IP 上に実装された GIOP を用いることにする。GIOP は、CORBA オブジェクト間のメッセージングに関する規定であり、データの符号化として CDR（Common Data Representation）を利用したものである。特に TCP/IP 上に実装された GIOP は、IIOP と呼ばれている。IIOP では、1 つの要求を複数のメッセージに分割して通信する手段も定義されているが、IIOP-1.0 ではこの機能は定義されておらず、非常に単純な方法のみが定義されている。CORBA の仕様では、すべての CORBA ライブラリは下位互換を義務付けているため、IIOP-1.0 を用いることで他のほとんどの CORBA ライブラリを用いた CORBA オブジェクトで通信可能である。また、複数の CORBA オブジェクトを同一プロセスで起動している場合には、本来、オブジェクト間の通信もデータの符号化も必要がない。そこで、基本方針（C） に従って、RtORB では、CORBA オブジェクト間の通信には、IIOP-1.0 を用い、同一プロセス内のオブジェクト間では、単純な関数呼び出しを用いるようにする。

CORBA オブジェクト間の通信に関して、GIOP ではデータの符号化として CDR が用いられていることを述べた。CORBA の仕様書では、CDR に関しても様々な定義がなされている。しかしながら、ロボットのソフトウェアを実装するにあたり、プログラム言語でサポートしているデータ型に関する符号化処理をすべて実装することは、効率が悪い。また、RT ミドルウェアでは、ロボットのソフトウェア開発時に必要なデータ型のほとんどを予め定義している。そこで、RtORB では、RT ミドルウェアで使用するデータ型を中心にほとんど利用されない Fixed 型などのデータ型に関する符号化処理の実装を行わないとする。これが基本設計(D)となる。

現在広く普及している CORBA ライブラリの多くは、UNIX 系または Windows のような比較的大規模かつ高機能な計算機ようのオペレーティング

システムをターゲットに実装されている。そのため、CORBA オブジェクト間の通信に関してスレッドの利用を基本としており、多用されている。スレッドは、本来、軽量プロセスでありプログラム開発において容易に並行処理を実現する機能である。しかし、このスレッドを多用すると多くの計算資源を消耗することになる。特に、分散オブジェクト間の通信処理においてスレッドは非常によく利用されている。また、組み込み向けの CPU や軽量のオペレーティングシステムでは、スレッドそのものをサポートしていないものもある。そこで RtORB では、基本方針（E）に従って、割り込み処理で実現できる機能は可能な限りスレッドを用いない実装をすすめる。これによって、CORBA オブエクト間の実装方法に制限が加わる場合があるが、RT ミドルウェアの API レベルでこの問題を回避し、知能モジュールの開発者には、ソースコードレベルでの互換性を確保する。

RtORB の実装

前述の基本設計に基づき RtORB の実装を行った。RtORB の動作のターゲットは、汎用のオペレーティングシステムのみならず、組み込み向けの小型のオペレーティングシステムも含める。そのため、C++言語の処理系を持たなオペレーティングシステムもあることから、実装言語として C 言語を用いることとする。

CORBA は、本来、オブジェクト指向言語を持ちて実装されることを前提としている。そのため、C 言語で実装されたライブラリは非常に少ない。また、CORBA サーバへのオペレーション要求を定義するインタフェースは、IDL コンパイラによって生成されるものであり、CORBA の実装レベルでの表現は、この IDL コンパイラの出力と密接な関係がある。

RtORB では、IDL コンパイラとして ORBit2 (http://projects.gnome.org/ORBit2/) で実装さたものを使うことにする。ORBit2 は、GNOME というデスクトップ環境を実現するライブラリ群の一部として開発されたものである。また、ORBit2 は、OMG の CORBA の C 言語のリファレンス実装になっているため、RtORB でもオブジェクトの構造に関しては ORBit2 の実装と非常に近いものになっている。

また、前述の基本設計から RtORB では、CORBA サーバーアプリケーションを管理する機能として POA のみの実装し、RootPOA のみを取り扱うことにしている。そのため、RtORB の内部では、CORBA の初期化時に ORB と RootPOA の 2 つのオブジェクトを生成し、他の CORBA オブジェクトの管理テーブルの持つ構造体として実装を行なっている。

RtORB における CORBA オブジェクトは、IDL コンパイラによって、その実装のひな形が出力されるようになっている。例えは、下記のように IDL において Echo オブジェクトが

```
interface Echo {
  string echoString(in string mesg);
};
```

定義された場合には、IDL コンパイラによって、以下のコードが生成される。

```
CORBA_Echo   impl_Echo__create(PortableServer_POA poa,
                CORBA_Environment *ev)  {
    CORBA_Echo retval;
    impl_POA_Echo *newservant;
    PortableServer_ObjectId objid;
    Newservant  =  (impl_POA_Echo  *)RtORB_calloc(1,sizeof(impl_POA_Echo),
                                                  "create...");
    newservant->servant.vepv = &impl_Echo_vepv;
    newservant->poa = poa;
    POA_Echo__init((PortableServer_Servant)newservant, ev);
    /* Before servant is going to be activated all
          private attributes must be initialized.  */
    /* ------ init private attributes here ------ */
    /* ------ ---------- end ------------- ------ */
    objid = PortableServer_POA_activate_object(poa, newservant, ev);
    RtORB_free(objid, " objid");
    retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);
    return retval;
}
```

この構造体が RtORB における CORBA オブジェクトとなり、実際のオペレーションをこの中に定義する。
また、RtORB における CORBA オブジェクト間の通信の概要を下図に示す。

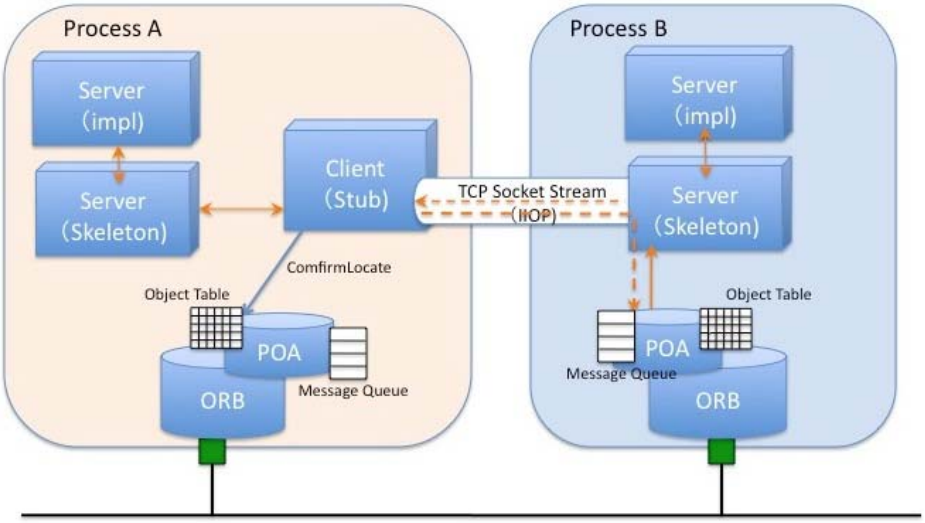

図 88　RtORB における CORBA オブジェクト間の通信

この図に示すように、RtORB では、ORB 部分は、自分のプロセス内の CORBA オブジェクト管理テーブルで管理しており、内部にあれば実装関数をコールし、そうでなければ、IIOP で対象 CORBA オブジェクトと通信するようになっている。また、各 CORBA オブジェクトはスレッドを生成せずに ORB のイベントループで通信を制御されている。

RtORB の詳細な実装と振舞に関しては、RtORB の配布パッケージおよび開発ホームページに記載している。

RT ミドルウェアに利用するための機能

前述のように RT ミドルウェアの軽量化のために、RtORB を C 言語で実装を行った。しかしながら、RT ミドルウェアは、C++言語での実装が行われており、実際に RtORB を RT ミドルウェアで使用するためには、RtORB のオブジェクトを C++のオブジェクトとの互換性を確保するために、RtORB のオブジェクトおよび関数等の置き換えを行う必要がある。そこで、RT ミドルウェアで利用されている API を中心に C++言語へのラッパの開発を実施した。
さらに、IDL コンパイラにおいても IDL から生成されるスタブ、スケルトン、実装のテンプレートにおいても外部の C++言語で実装されたオブジェクトを利用可能にするような機能拡張を行なっている。そのため RT ミドルウェアで RtORB を利用するためのコード変更は、非常に軽微なもので実現されている。
RtORB のラッパは、OpenRTM-aist-1.0 及び OpenRTM-aist-1.1 に対応しており、OpenRTM-aist-1.1 以降では、正規のソースコードに統合されている。

まとめ

RT ミドルウェアを軽量化するアプローチとして、通信レイアで用いられている CORBA ライブラリの軽量化をはかり RtORB の実装を行った。RtORB では、IDL コンパイラ以外は、ANSI-C 準拠の機能のみを利用し、他のライブラリ等も使用していない。そのため、組込 MPU などへの CORBA ライブラリとしても利用可能になっている。また、RtORB の軽量化を評価するために、2 つの単純なデータ交換を行うコンポーネントを作成し、omniORB4 を用いた正規の OpenRTM-aist-1.0 との比較を実施したところ、通信速度は約 4 倍高速になり RT コンポーネントのフットプリントも約 1/2 に軽減することができた。

また、この RtORB を用いた RT ミドルウェアの実装は、Android OS や T-KERNEL といった組み込み向けのオペレーティングシステムへの展開に利用されている。

(B) ネイティイブの通信媒体を直接利用した軽量化

PIC のようなリソースの乏しい CPU 上でも動作する RT コンポーネントを実現するために、RT コンポーネントの実行コンテキストのみをターゲット上で動作させ、より高速なホスト上で動作するプロキシコンポーネントと協調動作を行うことで通常の RT コンポーネントと同様のオブジェクトモデルを実現した軽量版 RT ミドルウェア　eRTC（Embedded RTC）の開発を実施した。

平成 20 年度には、CORBA に依存せずネイティブの通信媒体を直接利用した軽量化 RT ミドルウェア eRTC の基本設計を行った。その結果、軽量 CORBA による実装との差異があまり大きくないことが確認され、この eRTC の開発は(A)軽量 CORBA 実装による軽量化への統合することにした。また、リリース予定の OpenRTM-aist-1.0 の設計の見直しにより柔軟な通信方式を選択可能になったために、eRTC の開発で実施予定であった機能は、十分に実件できることが確認された。

(C) 組込み用途向け RT ミドルウェアの開発

前述の軽量 CORBA ライブラリ RtORB を用いると OpenRTM-aist を比較的資源の少ない MPU 上で動作させることが可能であることを述べた。しかしながら、この方法で実装された RT ミドルウェアの大部分についてはＣ＋＋言語の従来の実装を利用しているため、これ以上の軽量化は困難である。一方で、既存の組込み MPU の大部分では、そのリソース不足のため OpenRTM-aist をそのまま動作させることは困難であり、RTC-Lite や

RTC-CANopen、あるいは mini/microRTC など、ホスト PC を利用したヘテロジニアスな構成をとらざるを得ないのが現状である。そこで、OpenRTM-aist の全体を C 言語で実装することで、従来の動的な特徴を極力減らし、より一層の軽量化を目指した RT ミドルウェア OpenRTM-aist-C の実装を行った。これにより、一般的な CPU から組込み MPU デバイスまでを RT コンポーネントとして統一的に扱うことが期待できる。以下は、OpenRTM-aist-C の実装のための基本設計と実装方法について述べ、OpenRTM-aist-C を用いた RT コンポーネントの開発方法とその制限等について述べる。

OpenRTM-aist-C の基本設計

RT ミドルウェアを用いたシステムでは、ロボットの機能要素を共通の仕様に基づいたソフトウェアのモジュール化することで、各機能コンポーネントの再利用性を実現している。この共通仕様は、OMG で「Robotic Technology Component Specification ver.1.0」（以下、仕様 RTC-1.0 と呼ぶ）として標準化されている。この共通仕様では、コンポーネントの基本構造が定義されているのみであるが、既存の RT ミドルウェアとの相互利用性を考慮するとコンポーネント間の通信規約についても OpenRTM-aist と共通化を行う必要がある。そこで、OpenRTM-aist-C では、下記のような基本設計を行った。

- ✧ RT コンポーネント間の通信には、RtORB を利用する
- ✧ RT コンポーネントは、原則として仕様 RTC-1.0 で規定された IDL に準拠する
- ✧ 省資源の MPU 上での動作を可能とするため、データポートのみを実装する
- ✧ サポートするデータ型は、RT ミドルウェアの基本データ型のみとする

OpenRTM-aist-C の実装

前述の基本設計に従って、OpenRTM-aist-C の実装を行った。OpenRTM-aist-C の実装においては、仕様 RTC-1.0 で定義されている IDL の定義ファイルを、RtORB の IDL コンパイラを適用することで得られた CORBA サーバのテンプレートに対して、RT ミドルウェアの API で使用している関数部分について実装を行った。

以下に、仕様 RTC-1.0 で定義された IDL のインタフェースの中で OpenRTM-aist-C で実装された部分を示す。

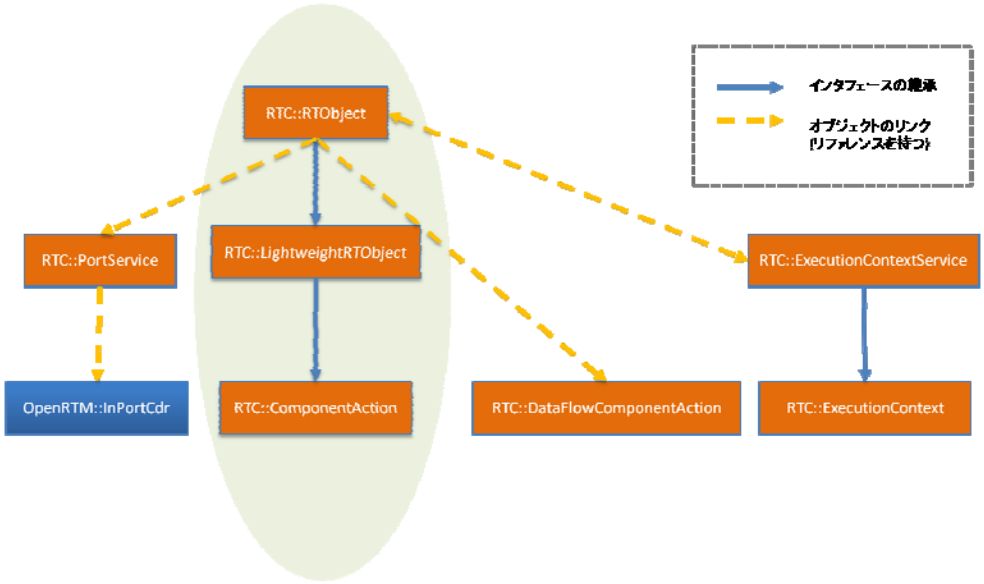

図 89 OpenRTM-aist-C のインタフェース構成（実装部分のみ）

「青矢印」がインタフェースの継承を意味し、「黄点線矢印」は継承はないが、インタフェースのオブジェクトリファレンスを持つことで、間接的にアクセス可能という意味を表している。
OpenRTM-ais-C において、RT コンポーネントのベースクラスは、「RTObject」インタフェースとし、継承元の「LightweigtRTObject」「ComponentAction」の機能をそのまま持つように実装されている。また、周期的 Activity を含む「DataFlowComponentAction」については、その CORBA オブジェクトのリファレンスを「RTObject」の CORBA オブジェクトに持たせることで、間接的に内部では継承関係を実現している。
さらに、実行コンテキスト用「ExecutionContextService」、Port 管理用「PortService」に関しても同様の実装方法がとられている。データ送受信用の「InPortCdr」は Port 接続時に生成管理させるため、「PortService」に関連づけを行っているが、OutPortCdr については、取り扱いがない。
なお、OpenRTM-aist-C は、軽量化を目指した最低限実装のため、OpenRTM-aist から若干構成変更を行っているが、共通性などに影響が出ないような実装を行っている。

次に、OpenRTM-aist-C を構成する全体のディレクトリ/ファイルを分割し、

概要について述べる。

- ディレクトリ構成

OpenRTM-aist-C のソースコードを展開すると表 18 のようなディレクトリが生成される。これらのディレクトリには、メイン実装部のヘッダファイルおよび C ソースファイル群があり、正規版の RT ミドルウェアと同様にインタフェース定義ファイルである IDL 群も含まれている。
また、OpenRTM-aist-C では、通信ライブラリとして RtORB を用いているために、 そのソースのすべてを包含した構成となっている。

表 18 ディレクトリ構成一覧

ディレクトリ	ディレクトリ説明
idl-compiler/	RtORB 付属の IDL コンパイラ。当システムコンパイル時にプログラムが生成され、 以降の IDL コンパイルの処理で使用される。(???.idl ファイルから[.h/.c]を生成するため)
CosName/	RtORB 付属の軽量化された NameServer。使用/未使用は任意で、本家の omniORB の NameServer を利用してもよい。
example/	本システム付属の RTC サンプル。[SimpleIO(単純データ送受信用)][SeqIO(可変長データ型送受信用)][MyDataIO(ユーザ定義データ型送受信用)]の 3 種類がある。
template/	本システムで動作する RTC 用のテンプレートファイル。
include/	本システムの各種ヘッダファイル群。
include/RtORB/	本システムで使用する[RtORB]ソースのヘッダファイル群。
include/rtm/	本システム[OpenRTM-aist-C]のメイン処理部のヘッダファイル群。
lib/	本システムの各種 C ソースファイル群。
lib/orb/	本システムで使用する[RtORB]ソースの C ソースファイル群。
lib/rtm/	本システム[OpenRTM-aist-C]のメイン処理部の C ソースファイル群。
lib/rtm/idl/	CORBA のインタフェース定義ファイル「IDL」群。
lib/rtm/impl/	IDL コンパイラで、インタフェース定義の IDL 群をコンパイルして抽出される、 「-skelimpl.c」において、1 ファイルのサイズが大きい事から、インタフェース毎に複数の ファイルに分割した際の、各種分割 C ソースファイル群。

- インタフェース定義ファイル（lib/rtm/idl）

表 19 に示されたファイル群は、OpenRTM-aist-C で用いられているインタフェース定義である。OpenRTM-aist-C では、基本的に OpenRTM-aist で使用しているが、「データ型」の拡張版を扱う「ExtendedDataTypes.idl」「InterfaceDataTypes.idl」の 2 つの IDL ファイルは組み込んでいない。

表 19　IDL ファイル(lib/rtm/idl)構成一覧表

ファイル名	ファイル説明
OpenRTM-aist.idl	以下の各種 IDL ファイルを include している。IDL コンパイラでは、当ファイルがコンパイルされる。
BasicDataType.idl	データ送受信などで使用する、デフォルトのデータ構造の型を定義したもの。（本家の IDL に対して、[TimedWChar] [TimedWString] [TimedWCharSeq] [TimedWStringSeq] struct 定義を削除。(使用不可のデータ型とする。)
DataPort.idl	データ送受信用の[DataPort]の処理に関して定義したもの。
Manager.idl	CORBA_Manager インタフェースを定義したもの。（実際には、中身の機能はほとんど未使用。）
OpenRTM.idl	【未使用】
RTC.idl	RTC に関する、主要インタフェースやパラメータが定義されている、メイン IDL ファイル。(本家の IDL に対して、DataFlowComponent インタフェースの継承変更あり。)
SDOPackage.idl	SDO 用 IDL。（内部的に、他の IDL で使用されている部分のみ必要）

- ヘッダファイル(include/rtm)

表 20 に示したファイルは、OpenRTM-aist-C で用いているヘッダファイルである。このファイル群は、各種 C ソースファイルにて、include するために定義情報をまとめたヘッダファイルと、機能毎に分割された関数群のヘッダファイルから構成されている。

表 20　ヘッダファイル(include/rtm)構成一覧表

ファイル名	ファイル説明
OpenRTM.h	各種ヘッダファイルの include をしており、また、「RTC_RtcBase」の define がある。 各 C ソースファイルから include されるメインヘッダファイル。 (ユーザの RTC からは、当ファイルのみを include すればよい。)

ファイル名	ファイル説明
DefaultConfig.h	RTC 自体のコンフィグ情報用に、デフォルトで用意されたパラメータ群。(本家 C++版の「DefaultConfiguration.h」「version.h」の中身を一つにまとめ、 さらに、Web マニュアルから必要項目を追加したもの。)
typedefs.h	各種データ型の typedef。「RTC_?????」と定義。
rtm-defs.h	共通データの定義用ファイル。
RTComp.h	RTC の各種 Activity のコールバック用の関数ポインタと、内部変数用のポインタを用意したもの。 (RTC の create()内で生成時にセットされ、Activity 関数から callback される時に使用される。)
Manager.h	Manager 関連関数用のヘッダファイル。Manager 構造体の宣言もあり。
RTObject.h	RTObject 関連関数用のヘッダファイル。
NamingManager.h	NamingManager 関連関数用のヘッダファイル。NamingManager 構造体の宣言もあり。
ExecutionContext.h	ExecutionContext 関連関数用のヘッダファイル。状態遷移用の構造体の宣言もあり。 (本家 C++ 版の「PeriodicExecutionContext」「StateMachine」クラスの機能に相当。)
Port.h	Port 関連関数用のヘッダファイル。Port 管理やデータ管理用の構造体の宣言もあり。 (本家 C++版の「PorAdmin」「PortBase」クラスの機能に相当。)
Properties.h	RTC のコンフィグレーション管理用関数のヘッダファイル。管理用データ構造体の宣言もあり。
Utils.h	各種文字列操作用 Utility 関数のヘッダファイル。 文字列管理用データ構造体の宣言もあり。
NVUtil.h	NameValue 型データの処理を扱う Utility 関数のヘッダファイル。

- Ｃソースファイル(lib/rtm)

  次に、OpenRTM-aist-C 本体のソースファイル群について説明する。OpenRTM-aist-C のソースファイルは、lib/rtm の下に格納さている。これらの中身の実装は、OpenRTM-aist-C の処理の流れと一致するように実装を行っている。 ただし、軽量化のためデータポートを介した通信機能を

実現する部分のみのコーディングが行われており、それ以外の部分は、空処理とした。
また、「OpenRTM-aist-?????」と名前付けられた「.h/.c」ファイルがいくつか存在するが、これらは、RtORB の IDL コンパイラにより、 自動生成されたファイル群である。 このファイルは、CORBA の通信を行う部分をブラックボックス化してくれている箇所であり、RT コンポーネントの開発者が手を入れる部分ではない。
ただし、「OpenRTM-aist-skelimpl」ファイルに関しては、IDL で定義されたインタフェースのメソッド部分の中身を実装する必要があるため、加筆している。なお、「OpenRTM-aist-skelimpl.c」ファイルについては、初期状態では全インタフェース情報が出力されておりソースコードが非常にが長く可読性が悪い。そのため、次節で説明するように、実装コードは複数ファイルに分割されている。表 21 に、各ファイルの一覧を示す。

表 21　C ソースファイル(lib/rtm)構成一覧表

ファイル名	ファイル説明
Manager.c	RTC の各種管理を行う、Manager 関連関数の実装ファイル。
RTObject.c	RTObject 関連関数の実装ファイル。 (RTC の生成(RTObject 型オブジェクト)/削除、RTC コンフィグレーションのセット/取得、 Port の生成/削除、データ送受信などの処理。) (ユーザの RTC からは、当ファイル内の関数を call する。)
NamingManager.c	NamingManager 関連関数の実装ファイル。(NameServer 名取得、登録用の RTC データ生成、RTC の登録/解除などの処理。)
ExecutionContext.c	ExecutionContext 関連関数の実装ファイル。 (EC 用スレッド生成、RTC との相互リンク、RTC の状態遷移の管理などの処理。)
Port.c	Port 関連関数の実装ファイル。 (Port 生成/削除、PortService オブジェクト生成/削除、PortProfile セット/取得、) 可変長データ型のセット/取得、データ送受信の Marshaling/Unmarshalin などの処理。)
Properties.c	RTC のコンフィグレーション管理用関数の実装ファイル。([RTC_Properties]型データの生成/削除/セット/追加/取得/

ファイル名	ファイル説明
	コピー/検索などの処理。)
Utils.c	各種文字列操作用 Utility 関数の実装ファイル。(区切り文字による、文字列の分割と、そのデータ格納などの処理。)
NVUtil.c	NameValue型データの処理を扱うUtility関数の実装ファイル。(NameValue 型データの生成/削除/セット/取得/コピーなどの処理。)
OpenRTM-aist.h	[IDL コンパイラが生成] 以降のファイルのヘッダ部。 (define/typedef、関数宣言、インタフェースやパラメータの宣言など。)
OpenRTM-aist-decls.h	[IDL コンパイラが生成] 各種内部フラグの定義用。
OpenRTM-aist-skelimpl.h	[IDL コンパイラが生成] IDL で定義されたインタフェース部実装用のヘッダファイル。
OpenRTM-aist-common.c	[IDL コンパイラが生成] CORBA 通信先関数 call 用の共通処理用ファイル。 (操作関数、付随する各種データの定義、TypeCode データの定義など。)
OpenRTM-aist-stubs.c	[IDL コンパイラが生成] CORBA のスタブ機能に相当。
OpenRTM-aist-skels.c	[IDL コンパイラが生成] CORBA のスケルトン機能に相当。
OpenRTM-aist-skelimpl.c	[IDL コンパイラが生成] IDL で定義されたインタフェース部実装用の C ソースファイル。

- C ソースファイル(lib/rtm/impl)

  上述のようにこのディレクトリに格納されているファイルは、OpenRTM-aist-skelimpl.c を各インタフェース単位で分割した後の実装コードである。これらのファイルは、OpenRTM-aist-skelimpl.c にインクルードされている。OpenRTM-aist-C の実装コードは、「**Impl_モジュール名_インタフェース名.c** 」としている。

  OpenRTM-aist-C では、データポートを利用したＲＴコンポーネントの機能を実現する部分と RT システムエディタ (RTSE)で操作する上で必須となるメソッド部分に関する実装だけを行っているものがある。表 22 に、各ファイルの一覧を示す。

表 22　C ソースファイル(lib/rtm/impl)構成一覧表

ファイル名	ファイル説明
Impl_OpenRTM_DataFlowComponent.c	【未使用】
Impl_OpenRTM_ExtTrigExecutionContextService.c	【未使用】
Impl_OpenRTM_InPortCdr.c	InPort 側でのデータ受信時の処理を実装したもの。
Impl_OpenRTM_OutPortCdr.c	【未使用】 (DataPort 送信は”PUSH”型のみのため。当ファイルは”PULL”型用)
Impl_RTM_Manager.c	【未使用】(ManagerServant を現在未使用なため不要)
Impl_RTC_ComponentAction.c	RTC の各種 Activity(状態遷移時の処理)の内部処理を実装したもの。
Impl_RTC_LightweightRTObject.c	RTC のメイン処理部で、初期化/終了処理などの処理を実装したもの。
Impl_RTC_ExecutionContext.c	ExecutionContext の各種操作用メソッドの実装と、パラメータのセット/取得用のメソッドを実装したもの。
Impl_RTC_DataFlowComponentAction.c	RTC の Activity の内、「on_execute()」「on_state_update()」「on_rate_changed()」の 3 メソッドに関して実装したもの。
Impl_RTC_DataFlowComponent.c	【未使用】
Impl_RTC_PortService.c	DataPort の接続/切断に関するメソッドと、Port 情報/接続情報などの 取得用のメソッドを実装したもの。
Impl_RTC_ExecutionContextService.c	ExecutionContext のオブジェクト生成と、 RTSE での内部処理対応のメソッドのを実装したもの。
Impl_RTC_RTObject.c	[LightweightRTObject]を継承し、さらに、RTC のプロファイル情報の 取得処理用のメソッドを実装したもの。
Impl_SDOPackage_SDOSystemElement.c	【限定使用】 (RTSE での内部処理対応でのみ使用)
Impl_SDOPackage_SDO.c	【限定使用】 (RTSE での内部処理対応でのみ使用)

ファイル名	ファイル説明
Impl_SDOPackage_Configuration.c	【未使用】
Impl_SDOPackage_SDOService.c	【未使用】
Impl_SDOPackage_Monitoring.c	【未使用】
Impl_SDOPackage_Organization.c	【未使用】

OpenRTM-aist との相違点

OpenRTM-aist-C は、RT ミドルウェアの C 言語版の実装であり、ライブラリ「 libRTM.so 」として コンパイル時に生成される。RT コンポーネント開発、実行時には、このファイルを用いることで OpenRTM-aist と相互運用を実現することができる。しかしながら、組み込み向け MPU への対応のため、多くの簡略化と軽量化をはかっている。そのため、正規の RT ドルウェアとの機能の相違がある。以下に、OpenRTM-aist と OpenRTM-aist-C との機能の相違について述べる。

- 全体比較

OpenRTM-aist-C では、以下に示す機能が実装している。

- OpenRTM-aist.idlに記述されたインタフェースに準拠した内部処理の実装をしており、他の RTM 実装との連携が一部を除いて可能。
- 「rtc.conf」による、コンフィグレーションのセットが可能
- Manager による、RTC の管理機能が可能
- NameServer での RTC 管理ができるような登録/解除処理が可能
- ExecutionContext(以降、EC)により、RTC の状態遷移の管理が可能
- 一方向通信(push 型)のみ可能な DataPort を持ち、RTC 間で 1 対 1 の DataPort 接続をすることで、データの送受信が可能
- 可変長のデータ型、ユーザが定義する独自データ型でのデータ送受信が可能
- 一部のバージョンの RtSystemEditor などのツールでの操作が可能

また、OpenRTM-aist-C では、主に下記の機能が省略している

- Factory オブジェクトによる、RTC 生成/削除の管理機能
- ManagerServant オブジェクトによる、ローダブルモジュール管理機能
- ModuleManager オブジェクトによる、モジュール管理機能

- Connector オブジェクトによる、データ送受信の管理機能
- Publisher オブジェクトによる、データ送信の管理機能
- Provider/Consumer オブジェクトによる、データ送受信の管理機能
- Pull オブジェクトによる、データ送受信の管理機能
- Buffer オブジェクトによる、データ送受信の管理機能
- Listener/Callback オブジェクトによる、イベント同期時の実行機能
- ExtTrig オブジェクトによる、イベントドリブン EC の実行機能
- Composite オブジェクトによる、複合 RTC 管理機能
- Logger オブジェクトによる、ログ出力機能
- Timer オブジェクトによる、タイムスタンプ管理機能
- etc...

以上のように、大幅な省略を行っているので、シンプルな機能を持つシステムとなっている。現状は、Manager は一つのＲＴコンポーネントに対して割り当てられ、RT コンポーネントのオブジェクトと実行コンテキスト(EC)とは「１対１」を想定しているため、 Object/Module の管理をする機能や、生成削除の Factory などの機能は省略している。また、一つの RT コンポーネントの処理機能を分散オブジェクト化することもしていないため、データ送信などで Publisher/Provider/Consumer/Connector などのオブジェクトは使用せず、内部でシーケンシャル処理として実装している。

図 90～図 92 は、 OpenRTM-aist オブジェクトと、OpenRTM-aist-C のオブジェクトの比較 を表したものである。OpenRTM-aist にて実装されているクラス等を左側に抽出し、OpenRTM-aist-C に 関連しているものだけを「 楕円 」で囲み、「 黒矢印 」で表示し、右側にある OpenRTM-aist-C 側の 該当ファイル名とリンクさせて記載している。関連項目の中での未使用機能や、機能そのものを未使用のものについては、「 薄い透明色 」にて表示している。また、図 93 は OpenRTM-aist-C で未実装のオブジェクトを示したものである。

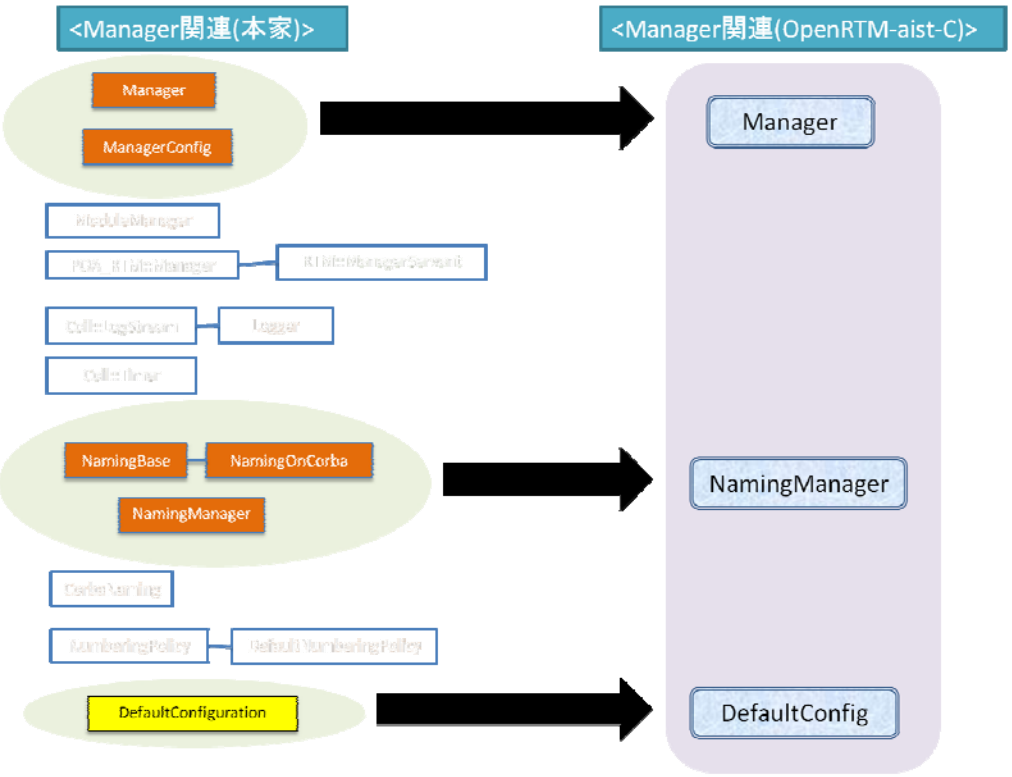

図 90　OpenRTM-aist オブジェクトと OpenRTM-aist-C のオブジェクトの比較（１）

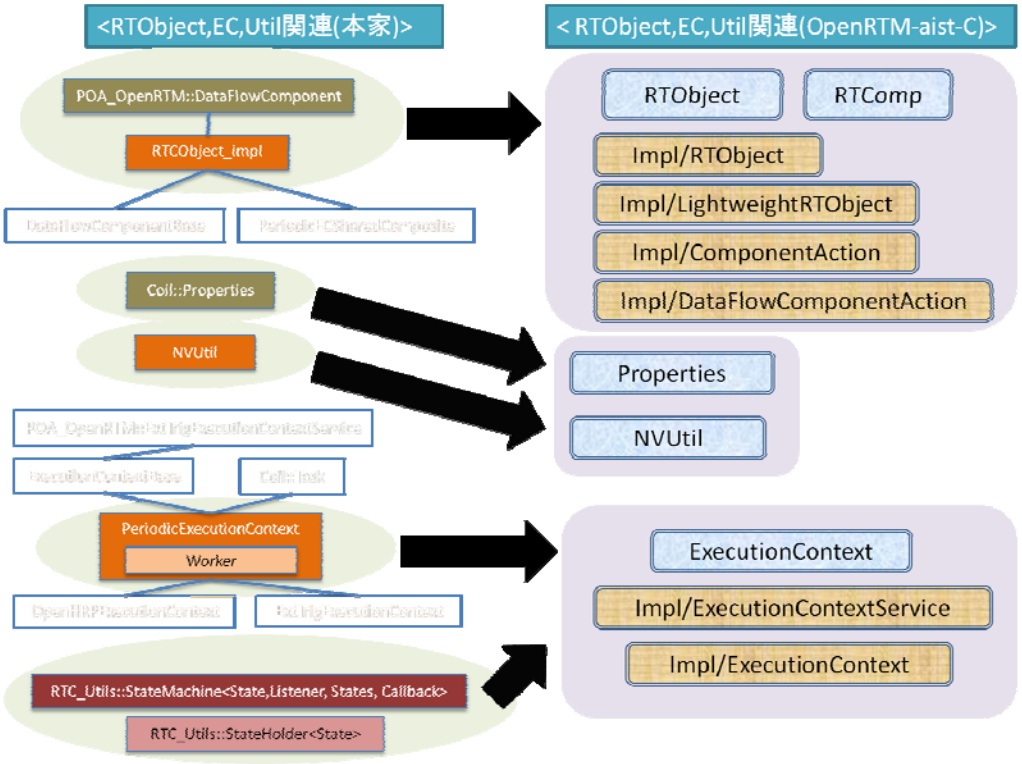

図 91　OpenRTM-aist オブジェクトと OpenRTM-aist-C のオブジェクトの比較（２）

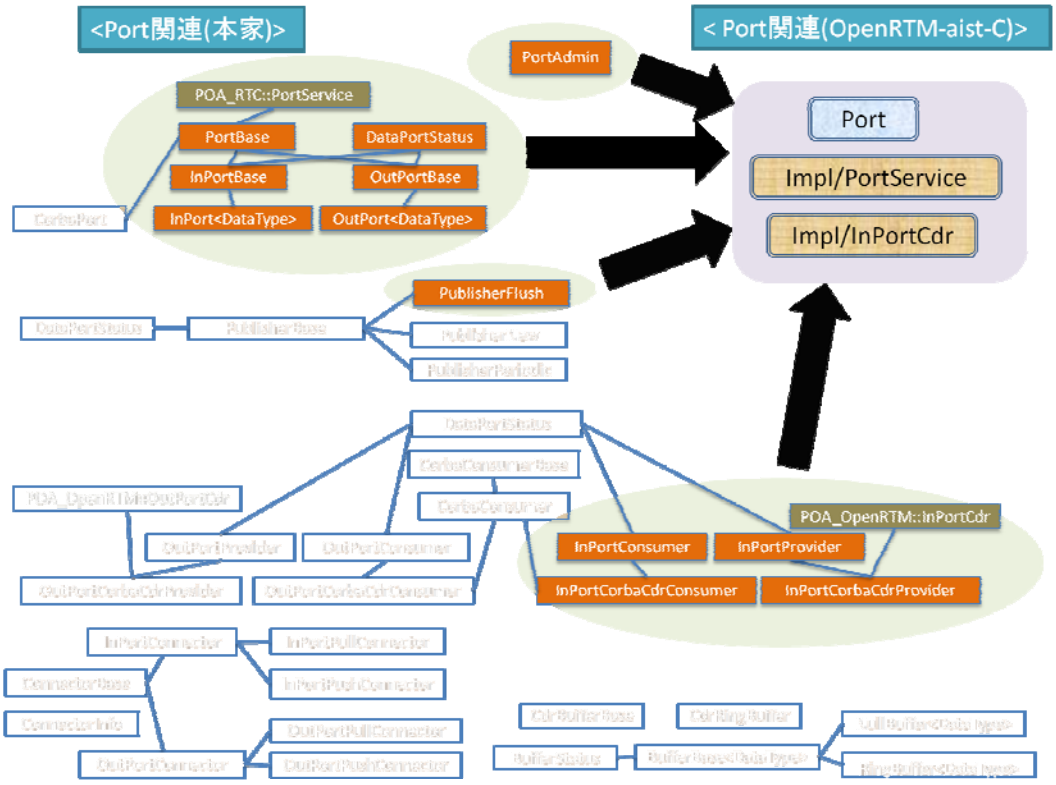

図 92　OpenRTM-aist オブジェクトと OpenRTM-aist-C のオブジェクトの比較(３)

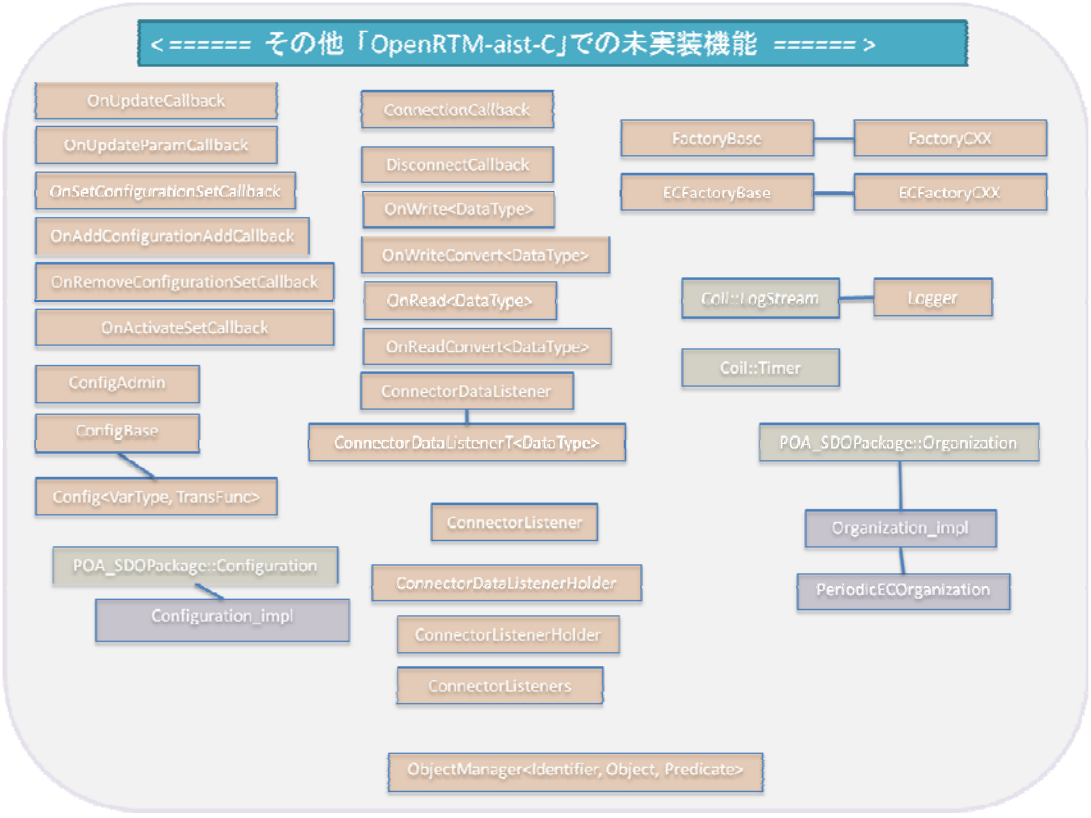

図 93　OpenRTM-aist-C で未実装のオブジェクト

まとめ

以上、OpenRTM-aist-1.0 の C 言語実装である OpenRTM-aist-C について述べた。この実装では、データポートを用いた RT コンポーネントのみを実装することで軽量化をはかっている。これは、組み込み向けの MPU 等では、その計算資源から多くの処理を同時に実行するよりも安定した周期実行の処理が望まれる場合が多く存在するためである。この OpenRTM-aist-C のソースコードとユーザマニュアル等の詳細ドキュメントに関しては、開発サイトに掲載を行っている。

(c-3) 高信頼ＲＴミドルウェアの開発

実世界で人とともに活動するロボットシステムを実現するためには、システムを構成する要素や部品の故障リスクを低減するための安全機能が必要不可欠である。次世代知能ロボットシステムの安全装置等の知能モジュールを実現するためには、機能安全規格に準じた開発プロセスを規定し、その開発プロセスに沿って知能モジュールを開発する必要がある。そこで、IEC61508 の機能安全の国際規格に基づいた開発プロセスを構築し、機能安全規格に準じた RT ミドルウェアの開発を行った。本研究項目の最終目標は、機能安全規格 IEC61508 に基づいた開発プロセス

を想定し、その開発プロセスを構築、支援するためのツール群を開発し、機能安全規格を取得済みの OS 上に IEC61508 準拠の RT ミドルウェアを実装することである。

成果の概要

生活支援分野など人と共存するサービスロボットにおいては、安心・安全のために、機能安全への対応が求められている。機能安全の国際標準規格として IEC61508 や ISO26262、ISO13482 などがあり、将来的には、サービスロボットはこれらの国際標準に準拠しなければならなくなると考えられる。一方、サービスロボットが市場に出て行くためには、ロボットの技術革新だけでなく、コストダウンも必要であるが、機能安全に対応するロボット開発は、その開発プロセスが複雑で、開発の難易度も高いため、どうしてもコスト高にならざるを得ない。また、現状の RT ミドルウェアはオープンソースのものだけであり、機能安全に対応したサービスロボットで利用できる高品質な商用の RT ミドルウェアは存在していなかった。

そこで、我々は、機能安全対応のためのコストダウン、高品質な RT ミドルウェア、さらに、安全モジュールの再利用による開発効率の向上のため、機能安全に対応した高信頼 RT ミドルウェア「RTMSafety」（仮称）の開発に取り組んだ。

RTMSafety の開発にあたっては、従来の研究開発におけるトライ＆エラーの開発やインクリメンタルな開発手法を採用することはできない。ウォーターフォール型のソフトウェア開発で用いられる V 字モデルの開発手法を取り入れるとともに、IEC61508 の機能安全対応の開発プロセスである全安全ライフサイクルに則って、RTMSafety の開発プロセスを策定し、RTMSafety を開発した。

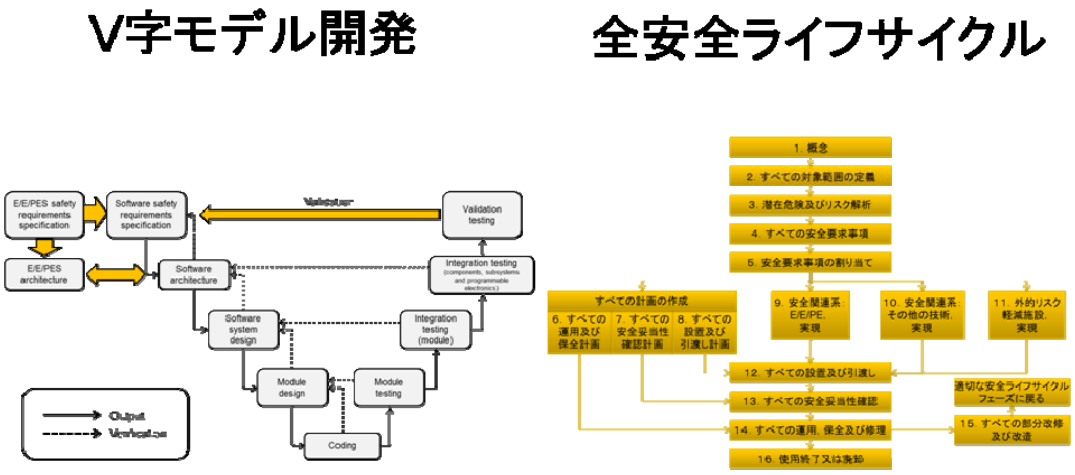

図 94　機能安全対応のシステムを開発するための開発プロセス