オペレーティング・システム 48－2
（１９９０．　９．　７）

実時間分散型プログラミング言語実現に向けて

石川 裕†、徳田 英幸‡
†電子技術総合研究所
‡カーネギー・メロン大学
yisikawa@etl.go.jp hxt@cs.cmu.edu

RTC++はC++を拡張した実時間制約を記述できるオブジェクト指向言語である。RTC++の特徴は以下の通りである。

1. リアルタイムオブジェクトと呼ばれるアクティブオブジェクトが定義できる。
2. アクティブオブジェクトをリモートホストに生成する機能がある。
3. ステートメントレベル、オブジェクトのメソッドレベルで時間制約の記述が可能である。
4. 周期的タスク記述能力がある。

本稿では、RTC++の言語機能の概要を述べた後、実時間(分散型)オブジェクト指向言語をARTS分散カーネルに実装するにあたって考慮した点について述べる。これを通して、実時間(分散型)オブジェクト指向言語を支援するために必要な機能を明らかにする。

Towards an Implementation of Distributed Object-Oriented Real-Time Language

Yutaka Ishikawa†, Hideyuki Tokuda‡
†Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba,
Ibaraki, 305, JAPAN
‡Carnegie Mellon University
Pittsburgh, PA 15213 USA
yisikawa@etl.go.jp, hxt@cs.cmu.edu, cwm@cs.cmu.edu

RTC++ is an extension of C++ and its features are to specify i) a real-time object which is an active entity, ii) creation of an active entity on a remote host, iii) timing constraints in an operation as well as in statements, and iv) a periodic task with rigid timing constraints. In this paper, we first describe an overview of RTC++ features. Then we present the experience of the implementaion of RTC++ on top of the ARTS kernel which is being develeped at CMU. We will make it clear what kernel functionalities or primitives are needed to support a distributed object-oriented real-time language.

1 はじめに

実時間分散処理はマルチメディアシステム、ロボット、航空機制御等で使われる重要な技術である。実時間という用語は単に高速処理応答性能という言葉に置き換わりがちである。これは、従来の計算機処理能力不足に起因するものである。最近の計算機処理能力の向上により、高速性よりも開発の容易性や保守性が重要になってきている。このような背景のなか実時間システムで最も重要なことは、時間的な制約 (timing constraint) を持つタスクがその制約を満たせるかどうかを静的に解析し、それが実機でも保証されるような技術の構築である。このような解析をスケジューラビリティ解析と呼んでいる。また、システムの状態をあらかじめ解析できるようなシステムを予測可能システム (Predictable System) と呼ぶ。

オブジェクト指向はプログラムのモジュール化を容易にする概念である。オブジェクト指向では、プログラムは複数のオブジェクトとそれらの間のメッセージ交換で定義される。概念的には、オブジェクトはメッセージを受け付けると対応するメンバ関数 (Smalltalk 用語ではメソッド) が呼ばれ、メンバ関数で記述されているプログラムが実行される。我々は、オブジェクトのメンバ関数毎に時間制約を付加することにより、論理的枠組だけでなく時間についてもオブジェクトでモジュール化することを提唱している [8]。これを"Timing Encapuslation" と呼んでいる。

RTC++は、この概念に基づいて設計された C++ を拡張したオブジェクト指向型実時間処理記述言語である。RTC++では、C++のオブジェクトに加えてアクティブオブジェクトと呼ばれる Threads of Control (以下単に Thread と呼ぶことにする) を持つオブジェクトを定義することができる。さらに、アクティブオブジェクトのメンバ関数に時間制約が記述されていた場合、これをリアルタイムオブジェクトと呼ぶことにしている。

リアルタイムオブジェクトと分散環境におけるリモートオブジェクト生成／交信を言語レベルで支援しているという意味において分散システム記述言語と呼ぶことにする。これは、ユーザが分散環境を意識してプログラムを記述する必要があることを暗に示している。

RTC++ は分散型実時間カーネル ARTS[8] 上で稼働すべく実装をしている。現在、RTC++と ARTS カーネルのインターフェイス部分の実装が終了している。RTC++コンパイラ[1]は、RTC++のソースコードを C++のソースコードと C のソースコードに変換する。本コンパイラは現在実装中であり、一部稼働している。

本稿では、RTC++の実行環境を ARTS カーネル上で実現するという体験をもとに、実時間分散型オブジェクト指向型言語を支援するために必要な OS 機能について考察する。また、従来型 OS で実現した場合の問題点についても触れる。

RTC++の設計思想や言語機能の詳細は文献 [4] を参照のこと。また、実時間処理に特有の最近の話題は文献 [7, 1] を参照のこと。

2 RTC++の概要

本節では、例題によって、RTC++のシンタックスとセマンティックスを示す。

2.1 リアルタイムオブジェクト

Threads を持つオブジェクトをアクティブオブジェクトと呼ぶ。アクティブオブジェクトで時間制約を持つオブジェクトをリアルタイムオブジェクトと呼ぶことにする。以下は RTC++におけるアクティブオブジェクトの定義例である。

```
active class Example1 {
private:
  char    buf[BUF_SIZE];
  int     count;
  int     background();
public:
  int     read(char* data, int size)
                    when(count > size);
  int     write(out char* data);
  int     open();
  int     close();
activity:
  slave   read(char*, int);
  slave   write(char*);
  slave   open(), close();
  master background() cycle(;;0t30m;);
}
```

C++のオブジェクト定義と違う点は、active というキーワードが class キーワードの前に付加されている点と、activity と呼ばれる部分がクラス定義中に

[1] AT&T の C++がコンパイラであるという意味では RTC++もコンパイラであると言えるが、C++と C に変換するトランスレータであるとも言える。

付加されている点である。ABCL/1[12], Concurrent Smalltalk[11], Orient84/K[2] などと違い、RTC++のオブジェクトは複数の Threads が存在する。activity 部ではこれを宣言している。activity 部を省略した場合は、単一 Thread となる。

上記の例では、アクティブオブジェクト Example1 が定義されている。Example1 オブジェクトは、read, write, open, close 操作（メンバ関数）が定義されている。

activity 部において記述されている以下の文は、一つの Thread が read メッセージを処理することを定義している。

```
slave  read(char*, int);
```

以下のように定義すると、5 つの Threads が read メッセージを処理することになる。

```
slave[5]  read(char*, int);
```

activity 部において、以下の記述は、バックグランド処理を定義するために用いられる。ここでは、30 分毎に background ルーチンが呼ばれるよう定義されている。

```
master background() cycle(;;0t30m;);
```

各メンバ関数には、以下のようにガード文が記述できる。

```
int  read(char* data, int size)
          when(count > size);
```

この例では、*count* > *size* が真である時に限り、read 関数が実行される。以下の例では、ガード文によりメッセージに対する実行が遅延された場合、busy ルーチンが呼ばれる。busy ルーチンが負の値を返すとアボートし、それ以外の値を返せばガード文が真になるまで実行が遅延される。

```
int  read(char* data, int size)
         when(count > size) onwait(busy());
```

以下はアクティブオブジェクトの生成例である。(1) では、実行しているプログラムと同一仮想アドレス空間上にアクティブオブジェクトを生成する。(2) では、新たに仮想アドレス空間を生成し、そこにアクティブオブジェクトを生成する。(3) では、host1 によって示されるリモートホスト上にオブジェクトを生成する。

```
Example1 *v;

v = new Example1 priority 4;                  // (1)
v = new Example1 at myhost priority 4;        // (2)
v = new Example1 at host1 priority 4;         // (3)
```

理想的には、ユーザは、アクティブオブジェクトが同一アドレス空間に存在するとかリモートホストに存在するとかということを気にしたくない。しかし、システムのスケジューラビリティを静的に解析するには、このような情報がなければ不可能である。

2.2 アクティブオブジェクト間交信とパラメータの受渡し

アクティブオブジェクト間交信は同期型のみを支援している。受信側オブジェクトは reply 文によって、送信側オブジェクトに値を返し、処理を続行することができる。

アクティブオブジェクト間のパラメータパッシングは、C++オブジェクトのパラメータパッシングとは異なる。RTC++[4] は、パラメータパッシングにおいてアドレス渡しを許していない。値は関数の返値としてしか受け取れない。しかし、ユーザは複数の値を返値で貰いたいであろう。そこで、新たにメンバ関数のアーギュメント定義部に in, inout, out という属性をつけることにした。例えば、先の Example1 クラスでは、write のアーギュメントは out 属性がついているので、write 関数で書き込んだデータは呼びだし側に反映される。

ユーザ定義のオブジェクトをアーギュメントとして授受する場合、そのオブジェクトに pack, unpack メンバ関数が定義されている必要がある。pack 関数はユーザオブジェクトをメッセージとして送れるよう変換し、unpack 関数はメッセージからユーザオブジェクトに変換するルーチンである。

例えば以下の例では、func1 関数は Example0 クラスのインスタンスをアーギュメントに持っている。

```
class Example0;
active class Example1 {
    ...;
public:
    int    func1(int a1, Example0 a2);
}

void
afunc()
{
    Example0    *a = new Example0;
    int         i, j;
```

```
    Example1    *ep;

    ...;
    ep = new Example1;
    j = ep->func1(i, a);
    ...;
}
```

以下に示すのが、RTC++が生成するCコードである。func1関数が呼ばれる前に、Example0のpackルーティンが呼ばれている。

```
void
afunc()
{
  Example0    *a = new Example0;
  int         i, j;
  OID     ep;
  char    buf[MAX_SIZE];
  char    *cp = buf;

  ...;
  rtcCreateObject(&ep, "Example1", 0);
  cp = intPack(cp, i);
  cp = a->pack(cp);
  j = rtcRequest(ep, EXAMPLE1_FUNC1, buf - cp, buf,
                 RVALUE_INT, 4);
}
```

2.3 時間制約プログラミング支援

以下は、リアルタイムオブジェクトの記述例である。readメンバ関数は、20msec以内に終らなければならないと定義されている。もし、20msec以内に終らなければread_abort関数が呼ばれる。

```
active class Example1 {
private:
  ...;
  int   read_abort();
  int   write_abort();
public:
  int   read(char* data, int size)
            when(count > size)
            within(0t20) timeout(read_abort());
  int   write(out char* data)
            within(0t20) timeout(write_abort());
  ...;
}
```

以下は、ステートメントレベルでの時間制約を記述するための構文である。timeはRTC++が提供するTimeクラスのインスタンスである。timeで示される時間以内に<within body>の実行が終了しなければ、<timeout body>が実行される。なお、within文以外に、before, after文がある。

```
within(time) {
    <within body>
} except {
case timeout:
    <timeout body>
}
```

以下は、周期的タスクを実現するための構文である。starttimeで示される時間からperiodで示される時間間隔の周期が始まり、endtimeで示される時間まで続く。各周期は、deadlineで示される時間以内に終了する必要がある。

```
cycle(starttime; endtime; period; deadline) {
    ...;
} except {
case timeout:
    ...;
}
```

2.4 クリティカルセクション

以下は、クリティカル・リージョンを実現するための構文である。rrはRegionクラスのインスタンスである。region内は、唯一つのThreadのみが実行可能である。

```
region (rr) {
    ...;
} except {
case abort:
    ...;
}
```

3 OS支援機能

ここでは、一般的なオブジェクト指向を支援する場合と実時間システムを支援する場合の2つの立場で、OSが支援すべき機能について議論する。

3.1 アクティブオブジェクト実現

アクティブオブジェクトを実現する際に、2つの方法がある。1つは、アクティブオブジェクト毎にThreadsを持たせる方法である。この場合、アクティブオブジェクトが生成される度に必ず一つ以上の実行スタック等の環境を生成する必要がある。これは、システム資源を消費するという意味で弱点を持っている。しかし、近年の主記憶の巨大化により、この弱点は致命的ではない。

他の方法は、Cloudsシステム[10]や京都大学のToMのようにオブジェクト間交信の際、スタックのような実行環境をそのまま流用していくことにより、カーネルが抱える実行環境を減らす方法がある。し

かしAdaのようなランデブの場合、ランデブの終了後2つのThreadに分かれることになる。これを実現するには、実行の途中でforkのような操作が必要となる。このようなプログラミングスタイルで記述されたプログラムの性能低下が考えられる。

予測可能実時間システムという立場からでは、後者は動的にforkされるのでシステム資源がどの程度必要になるのか静的に把握できない。前者は動的アクティブオブジェクト生成がない場合に静的に把握できるので良い。この理由により、ARTSカーネルは後者を採用している。

3.2 アクティブオブジェクトと仮想アドレス

アクティブオブジェクトと仮想アドレスについて考える。3つの方法が考えられる。一つは、全てのアクティブオブジェクトは別々の仮想アドレスを持つという方法である。これは、プロセスと同じ考えである。この場合、アクティブオブジェクト間で頻繁にメッセージ交信が行なわれるとコンテキスト変換のオーバヘッドの増大が予想される。

次の方法は、一つの仮想アドレスに全てのオブジェクトを格納する単一アドレス空間方式である。これは、現在のページ方式MMUの計算機アーキテクチャ上ではオブジェクト間のプロテクションが難しい。分散環境では分散共有メモリを実現する必要になる。これは、余分な同期機能が必要となり好ましくない。

最後は、一つのアドレス空間に複数のアクティブオブジェクトが存在するという方法である。この場合、頻繁にメッセージ交信が行なわれるオブジェクトは一つのアドレス空間に置くことができコンテキスト変換のオーバヘッド減少が期待される。この方法では先の理由により、一つのアドレス空間を分散環境で共有しない。この方法の弱点は、オブジェクト間でのプロテクションが支援できないという点である。しかし、このような最後の方法による機能をOSが提供しておけば、プログラミング言語設計者は自由にオブジェクト生成を制御できる。

ARTSカーネルは最後の方法を採用し、RTC++はこの機能をそのままユーザに提供している。先に述べたように、これによりユーザは最適なオブジェクト生成が可能となる。

3.3 オブジェクト間交信機能

ガード文をどのように実現するかという問題がある。一つは、カーネルからのupcallという方法がある。例えば、以下のようにメッセージ受信時にガード文に相当する関数のアドレスを渡す。

```
Recevie(&inv_dsc, &msg_dsc, &func);
```

関数はメッセージのアーギュメントの内容とオブジェクトの変数が参照できる必要がある。カーネルはメッセージキューの中から各メッセージをパラメータにして関数を呼びだし、その返値からメッセージをユーザに渡す。

この方法では、カーネルがユーザの関数を実行するためカーネルモードでは呼び出すことが出来ない。ユーザモードにしてユーザのコンテキストで実行しその結果を見てメッセージを選択しなければならないため、カーネル／ユーザのコンテキスト変換が起こってしまう。

他の方法は、カーネルは何もせずにオブジェクト側が全てのメッセージを取り込んで処理する方法である。この場合、実行環境においてメッセージを待ち行列に入れて管理する必要がある。RTC++の実現ではこの方法を採用している。アクティブオブジェクトが生成されると“root thread”と呼ばれるThreadが必ず一つ生成され以下の処理を行なう。

1. 必要なslave threadとmaster threadを生成する。
2. メッセージを受け付ける。
3. もしメッセージに対応するメンバ関数にガード文が定義されていれば、ガード文に対応する関数を呼び出し、その返値によりwaiting queueに入れるかslave threadが取り出すready queueに入れる。
4. 2に戻る。

このように、OSでオブジェクト指向言語を支援する機能としては、単純なオブジェクト間交信だけがあれば良い。従来のようなポート、ソケットというOSが提供している概念は必要ない。むしろ、そのような概念のもとでオブジェクト指向言語を実装しようとすると、効率の良い処理系はできなくなる。

例えば、ポートを支援しているようなOSの上でオブジェクト指向言語を実現することを考えてみよう。ポートに一つのメンバ関数のためのメッセージを対応するということが考えられる。しかし、一つのオブジェクトが持つメンバ関数は100を越える場合があり、これら全てにポートを割り当てるのは非現実的である。従って、一つのオブジェクトに一つのポートを割り当てることになる。するとオブジェクトに必要な実行環境を作るためには、Threadを生成するためのプリミティブとポートを生成するためのプリミティブを呼び出す必要がでてくるので、余分なプリミティブの呼び出しが生じる。

3.4 実時間支援機能

within, cycle文等を実現する方法について述べる。Unix等では、timerの設定と時間切れによるシグナルによってしかこのような実時間機能を実現できない。これでは、ネストしたwithin文などが実現できない。

また、Unixのような従来型OSで提供されているsleepプリミティブでcycle文を実現しようとすると以下のように実現されるであろうが、正しく動くとは限らない。

```
nexttime = time(0);
for(;;) {
    <cycle body>;
    nexttime += phase;
    curtime = time(0);                    /* (1) */
    delaytime = nexttime - curtime;       /* (2) */
    sleep(delaytime);
}
```

ここでは、cycle文の最後に次の周期までsleepするように計算している。しかし、(1)で現在時刻を取り出し(2)でsleep時間を計算するまでの間にプロセスがプリエンプションされた場合、delaytimeは正しく計算されなくなる。

RTC++/ARTSでは、以下のようにしている。ここでは、within文を例にするが、cycle文等も同様である。within文はコンパイラによって以下のようなswitch文に変換される。

```
swich(WithinPrimitive(time)) {
case NORMAIL:
    <within body>;
    break;
case TIMEOUT:
    <timeout body>;
    break;
case ABORT:
    <abort body>;
    break;
}
```

WithinPrimitiveは、Unixのsetjmp, longjmpに相当するような機能も含めている。カーネルは、timeに基づいてUnixのsleeping queueと同様のリストを生成し、NORMAL値を返す。このリストをtiming constraints listと呼んでいる。リストの要素には、timeoutとabortに備えてWithinPrimitiveが呼ばれる寸前のレジスタの状態が格納される。timing constaints listは、within, cycle, after, before, sleep等の実現のために使われる。クロック割り込み毎に起動されるカーネルのタイマルーチンは、先頭のリストの時間を減らしていき、0になるとそのリストの要素のタイプにしたがった操作を行なう。例えば、withinの場合はtimeoutが発生したことになるので、リストの要素に格納されているレジスタを使って、あたかもWithinPrimitiveからTIMEOUTという返値で戻ってきたようにユーザの環境を変える。

3.5 プライオリティ・インバージョン

複数のThreadsが共有資源を使う時に、プライオリティ・インバージョン(Priority Inversion)という問題が発生する[6, 9]。例えば3つのThreadsがそれぞれ異なる優先順位で走っていると仮定する。今、最も低い優先順位を持つThreadがクリティカルリージョンに入ったとする。この時、最も高い優先順位のThreadが実行可能となり同じクリティカルリージョンに入ろうとしても、すでに使われているために待ち状態になってしまう。中位の優先順位を持つThreadが走り出すと、低い優先順位のThreadは待ち状態のままになるので、結局高い優先順位のThreadは待ち状態になる。このように、高い優先順位のThreadが走れずに、それよりも低い優先順位のThreadが走ってしまう状況をプライオリティ・インバージョンと呼んでいる。この状況下では、各Threadの最大待ち時間を確定することができないので、スケジューラビリティ予測可能システムを構築することができない。

プライオリティ・インバージョンを解決する方法の一つとしてプライオリティ・インヘリタンスという方法がある[6]。先の例では、高い優先順位のThreadがクリティカルリージョンに入ろうとする時、低い

優先順位のThreadが使っているので、そのThreadの優先順位を高い優先順位と同じ優先順位に一時的に上げてしまう。こうすることによって、中位の優先順位のThreadは走らずに、低い優先順位のThreadがクリティカルリージョンを走り抜けていくことが可能となる。従って、高い優先順位のThreadの最大待ち時間を確定することができる。

複雑な場合は、クリティカルリージョンに入っている低い優先順位を持つThreadが、別のクリティカルリージョンに入ろう（ネストした場合）として待たされている時がある。この場合に、高い優先順位のThreadが外側のクリティカルリージョンに入ろうとした時に優先順位を伝搬していく必要がある。同様に、Threadが他のオブジェクトと交信していて待ち状態に入っている場合は、受信側オブジェクトの優先順位を上げる必要がある[5]。

ARTSカーネルV1.0では、オブジェクト交信で待ち状態に入っている場合の優先順位伝搬以外は実装されている。

ネットワークハンドリングにおけるプライオリティインバージョン問題に関する問題点とその解決方法は文献[9, 3]を参照のこと。文献[3]には、Unix4.3BSD流ネットワークハンドリングと我々の方法とをARTS上で実際に実装比較した結果が載っている。

4 まとめ

本稿では、分散型実時間オブジェクト指向言語の実装経験をもとに、実時間、オブジェクト指向の立場からOS機能に必要な機能について概観した。まとめると、以下のOS機能が必要である。

1. 同一仮想空間/別仮想空間内にオブジェクトを生成する機能
2. Multiple Threads of Control生成機能
3. 単純なオブジェクト-オブジェクト間交信機能
4. within/cycle等を支援する機能
5. クリティカルリージョンにおけるプライオリティ・インバージョンを回避する機能

ARTSカーネルは、Sun3、News両ワークステーションおよびFORCEボード上で稼働している。ARTSカーネルV1.0は、まもなくCMUから一般公開される予定である。

謝辞

本研究に関して、様々な議論をしていただいたCMUのARTグループの方々に感謝します。本研究遂行にあたっては、電子技術総合研究所の棟上昭男情報アーキテクチャ部長、二木厚吉言語システム室長の御支援に感謝いたします。

References

[1] ACM Operating Systems Review, Vol. 23, No. 3, July, 1989.

[2] Ishikawa, Y., Tokoro, M., "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation," Proceedings of OOPSLA-86, Portland, Sept. 1986, pp. 232-241.

[3] Ishikawa, Y., Tokuda, H., Mercer, C.W., "Priority Inversion in Network Protocol Module," Proceedings of 1989 National Conference of the Japan Society for Software Science and Technology, October, 1989.

[4] Ishikawa, Y., Tokuda, H., Mercer, C.W., "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints", To appear in OOPSLA90, 1990.

[5] Mercer, C. Tokuda, H., "The ARTS Real-Time Object Model," to appear in 11th IEEE Real-Time Systems Symposium, December, 1990.

[6] Sha, L. and Rajkumar, R., and Lehoczky, J. P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Carnegie Mellon University, CMU-CS-87-181, 1987.

[7] Stankovic, J.A., "Misconceptions abort Real-Time Computing: A Serious Problem for Next-Generation Systems," IEEE Computer, Vol. 21, No.10, October, 1988.

[8] Tokuda, H., Mercer, C.W. "ARTS: A Distributed Real-Time Kernel," Operating Systems Review, Vol.23, No.3, July, 1989, pp.29-53.

[9] Tokuda, H., Mercer, C. W., Ishikawa, Y. and Marchok, T. E., "Priority Inversions in Real-Time Communication," Proceedings of 10th IEEE Real-Time Systems Symposium, December, 1989.

[10] Wilkenloh, C. J., et. al., "The Clouds Experience: Building an Object-Based Distributed Operating System," USENIX Workshop proceedings of Distributed and Multiprocessor Systems, Fort Lauderdale, Oct. 5-6, 1989, pp. 333-347.

[11] Yokote, Y., Tokoro, M., "Experience and Evolution of ConcurrentSmalltalk," Proceedings of OOPSLA-87, Orland, Oct. 1987, pp.406-415.

[12] Yonezawa, A., Briot, J-P., Shibayama, E., "Object-Oriented Concurrent Programming in ABCL/1," Proceedings of OOPSLA-86, Orland, Oct. 1986, pp.258-268.