

実時間処理記述言語:RTC++

石川 裕†、徳田 英幸‡

†電子技術総合研究所

‡慶応大学 / 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++ 設計の基になったリアルタイムオブジェクトモデルを紹介し、スケジューリング解析の方法について述べる。次に RTC++ 言語機能の概要を述べる。最後に、今後の研究課題について述べる。

A Real-Time Programming Language: RTC++

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

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. The RTC++ is designed based on a realtime object model. In this paper, we first describe this model. Then, an overview of RTC++ features is described. Finally, the future works are mentioned.

1 はじめに

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 [6]。これを "Timing Encapsulation" と呼び、時間制約が付加されているオブジェクトをリアルタイムオブジェクトと呼んでいる。

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++ は分散型実時間カーネル ARTS [6] 上稼働している。

本稿では、まず、リアルタイムオブジェクトモデルの概要を述べる。本モデルによって実時間処理における静的スケジューリング可能性解析をどのように行なうか例を示しながら解説する。次に、RTC++ の言語仕様を例を示しながら紹介する。最後に、今後の研究課題について述べる。

2 リアルタイムオブジェクトモデル

2.1 単純なモデル

まず、単純なリアルタイムシステムモデルを考える。リアルタイムシステムは周期タスクとそれらが呼び出す並行オブジェクト [8] から構成されていると仮定する。並行オブジェクトは横取り不可能な単一スレッドから構成されているとする。すなわち、並行オブジェクトはメッセージを逐次に処理し、高い優先順位を持つメッセージが到着しても、現在の処理が終了するまで高い優先順位を持つメッセージは待たされる。優先順位逆転現象 (Priority Inversion) が起こらないように、優先順位継承 (Priority Inheritance) [5] によってオブジェクトの実行優先順位を一時的に上げる機構を導入する。

オブジェクト間の交信の図式を図 1 に示す。(2) で送信オブジェクトは受信オブジェクトにメッセージを送り返答を待つ。受信オブジェクトは (4) で返答メッセージを送り、処理を続ける (6)。送信オブジェクトは返答メッセージを貰うと処理を再開する (5)。このように (5) と (6) の部分は並行に処理が進められる。

ここでは、タスクの終了時間に制約があるような実時間処理記述について考える。あるタスクは幾つかのオブジェクトを呼び出すことによって実現されているので、それぞれのオブジェクトのメソッドにも時間情報として実行時間を付加すべきである。そこで、図 2 に示すような単純なリアルタイムオブジェクトモデルを定義する。単純リアルタイムモデルでは、メソッドの実行時間が表現されている。

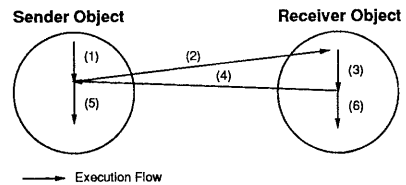


図 1: 単純なオブジェクトモデル

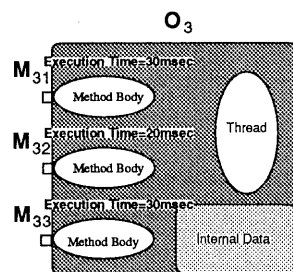


図 2: 単純なリアルタイムオブジェクトモデル

スケジューリング可能性解析のために以下のような仮定をする。あるメソッドが呼び出す他のオブジェクトのメソッドおよびその呼び出し回数は静的に決まる。再帰的呼び出しはない。単純リアルタイムオブジェクトにおけるオブジェクトの時間情報を表現するために以下のような記法を使う。

$Sm(o)$	オブジェクト o のメソッド集合
$C(m, o)$	オブジェクト o の持つメソッド m の実行時間
$Ms(m, o)$	オブジェクト o の持つメソッド m が呼び出す他のオブジェクトのメソッドのマルチ集合

単純リアルタイムオブジェクトモデルの例を図 3 にしめす。オブジェクト O_1 は、実行時間 50 msec のメソッド M_1 を持つ。オブジェクト O_2 は、実行時間 30 msec のメソッド M_2 を持つ。オブジェクト O_3 は、3 つのメソッド M_{31} , M_{32} , M_{33} を持ち、それらの実行時間は、それぞれ 30 msec, 20 msec, 30 msec である。図において、矢印はオブジェクトの呼び出し関係を表している。オブジェクト O_1 のメソッド M_1 はオブジェクト O_3 のメソッド M_{31} と M_{32} を呼び出し、オブジェクト O_2 のメソッド M_2 はオブジェクト O_3 のメソッド M_{32} を呼び出している。

これらの情報を使ってプログラムの時間制約を調べる。 O_1 の M_1 は O_3 の M_{31} と M_{32} を呼び出しているので、 M_1 の実行時間は M_{31} と M_{32} の実行時間の和よりも等しいか大きくなければならない。また、 M_2 の実行時間は M_{32} の実行時間よりも等しいか大きくなければならない。すなわち以下の式が満たされなければならない。

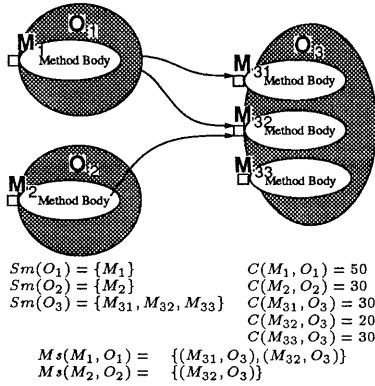


図 3: 単純リアルタイムオブジェクトモデルの例

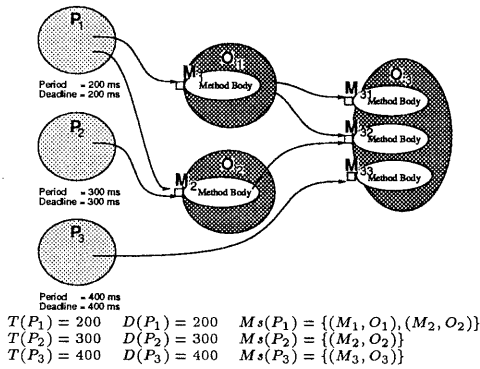


図 4: 単純リアルタイムオブジェクトモデル

$$C(M_1, O_1) \geq C(M_{31}, O_3) + C(M_{32}, O_3) \\ \rightarrow 50 = 30 + 20 \\ C(M_2, O_2) \geq C(M_{32}, O_3) \rightarrow 30 > 20$$

実行時間周期タスクは周期とデッドラインを持っている。タスクは複数のオブジェクトとの交信によって実現されるので、以下のように周期タスクを表現することが可能である。

$T(n)$	タスク n の周期
$D(n)$	タスク n のデッドライン
$Ms(n)$	タスク n が呼び出す他のオブジェクトのメソッドのマルチ集合

図 4 は、先に定義したオブジェクトを使って表現されている周期タスクの例である。RM(Rate Monotonic)スケジューリング [3] の基で、この例題のスケジューリング可能性を解析する。

RM スケジューリングとは、横取り可能な周期タスクに対する固定優先順位型スケジューリングであり、周期の短いタスクに高い優先順位を与えるというものである。タスク i の CPU 利用率 $U(i)$ を $U(i) = \frac{C(i)}{T(i)}$ で与える。こ

で、 $C(i)$ を実行時間、 $T(i)$ を周期とする。

周期の終りがタスクのデッドラインとした時、RM スケジューリングでは、以下の式が成立するならば全てのタスクはスケジューラブルであることが証明されている [3]。

$$\sum_{i=1}^n \frac{C(i)}{T(i)} \leq n(2^{1/n} - 1) \quad (1)$$

さらに、優先順位継承を採用している場合の RM スケジューリングでは以下の式が成立するならば全てのタスクはスケジューリング可能であることが証明されている [5]。

$$\frac{C(1)}{T(1)} + \dots + \frac{C(n)}{T(n)} + \max\left\{\frac{B(1)}{T(1)}, \dots, \frac{B(n-1)}{T(n-1)}\right\} \leq n(2^{1/n} - 1) \quad (2)$$

我々はこの結果を用いて図 4 のスケジューリング可能性を解析する。タスク P_1, P_2, P_3 の周期はこの順番に長くなっていく。したがって、RM スケジューリングにより、タスク P_1, P_2, P_3 の優先順位はそれぞれ、高、中、低となる。

まず、全てのタスクの実行時間について調べる。オブジェクトの全てのメソッドは実行時間が定義されているので、タスクの実行時間を見積もるのは簡単である。タスク P_1 は O_1 の M_1 (50 msec) と O_2 の M_2 (30 msec) を呼んでいるので、実行時間は 80 msec となる。 P_2 と P_3 の場合は、それぞれ 30 msec となる。

次に、最下位優先順位以外のタスクのブロック時間を調べる。 P_1 の実行が P_2 によってブロックされる場合は 2 つある。一つは、 P_2 が O_2 の M_2 を呼び出している時に、 P_1 が同じメソッドを呼び出す場合である。この場合の P_1 の最悪ブロック時間は 30 msec となる。なぜならば、最悪で P_1 はメソッド M_2 の実行を待たなければならないからである。2 番目の場合は、 P_2 が M_2 を呼び M_2 が O_3 の M_{32} を呼んでいる時に、 P_1 が O_1 を呼び O_1 が M_{31} または M_{32} を呼んでいる時である。 M_{31} と M_{32} の双方が P_2 によってブロックされることはないが、それらのうちのどちらかの実行はブロックされる。したがって、 O_3 の最悪ブロック時間は 20 msec となる。

P_1 は P_2 によって、 O_2 の実行時に 30 msec、 O_3 の実行時に 20 msec ブロックする可能性がある。 P_2 が実行中に P_1 が走り出した時にブロックする可能性の最大値が最悪ブロック時間となる。すなわち、30 msec である。

ブロック時間について P_1 と P_3 の関係について考える。 P_3 によって P_1 がブロックされる時は、 P_1 が O_3 の M_{31} または M_{32} を呼び出す時、すでに P_3 によって M_{33} が実行されている時である。この場合、 P_1 は最大 30 msec 待たされる。なぜならば、 O_3 の M_{33} の実行時間が 30 msec だからである。

P_1 のブロック時間をまとめると、 P_2 によって 30 msec、 P_3 によって 30 msec 待たされるので、全体で 60 msec 待たされることになる。同様に P_2 のブロック時間を求めると、30 msec となる。表 1 に時間情報の解析結果を示す。この表を使って RM スケジューリングにおけるス

表 1: 図 4 の時間情報 (単位 msec)

プロセス	周期 (T)	デッドライン	実行時間 (C)	C/T	ブロッキング時間 (B)	B/T
1	200	200	80	0.4	60	0.3
2	300	300	30	0.1	30	0.1
3	400	400	30	0.075	0	0

ケジューリング可能性解析を行なう。式 (2) と表から以下の式が導かれる。

$$\begin{aligned} & \frac{C(1)}{T(1)} + \frac{C(2)}{T(2)} + \frac{C(3)}{T(3)} + \max\left(\frac{B(1)}{T(1)}, \frac{B(2)}{T(2)}\right) \\ &= 0.4 + 0.1 + 0.075 + \max(0.3, 0.1) \\ &= 0.875 > 3(2^{1/3} - 1) = 0.780 \end{aligned}$$

これは、RM スケジューリングではスケジュール不可能であることを意味する。なお、次の節では、オブジェクト O_3 を変更して同一タスクセットがスケジューリング可能となることを示す。

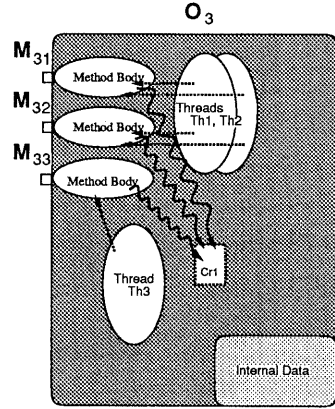
このように、オブジェクトに時間情報の仕様を入れることは以下のような利点がある：1) スケジューリング可能性解析が容易になる、2) リアルタイムオブジェクトを部品として再利用することが出来る。

2.2 リアルタイムオブジェクトモデル

先の例題では、オブジェクトの実行が逐次化されていたためにオブジェクトを呼び出しているタスクが待たされ、デッドラインが満たされなかった。そこで、待たされる時間（ブロッキング時間）を減らす方法について考える。これには、オブジェクト内並行実行と強制終了手法が考えられる。もしオブジェクト内で複数のスレッドが走り、それらがメソッドの実行を担当すれば、オブジェクトに要求されるメッセージは並行に処理することが可能となる。しかし、オブジェクトの内部データの同期によるブロッキング時間がある。

強制終了手法では、あるプロセスがオブジェクトにメッセージを送ってブロックした場合、そのオブジェクトの実行を強制終了させ、そのメッセージのための処理を行なわせる。実行の強制終了では、オブジェクトは内部状態の一貫性を保つための処理を行なわなければならない。当然、強制終了のための実行時間がブロッキング時間よりも小さくなければ、この手法は使えない。本論文では、紙面の関係で強制終了手法についてのモデルは提示せず、オブジェクト内並行実行モデルについて示す。

リアルタイムオブジェクトモデルは複数のスレッドを定義することが出来る [4]。各々のスレッドは一つのメソッドまたは幾つかのメソッド実行を受け持つ。幾つかのスレッドが幾つかのメソッド実行を受け持つことも出来る。このようなスレッドはスレッドグループと呼ばれている [1]。リアルタイムオブジェクトは以下のような情報によってモデル化される。



$T \rightarrow M$: Thread T is responsible for executing M

$M \rightsquigarrow Cr$: Method M accessing Critical Region Cr

$$\begin{aligned} Sm(O_3) &= \{M_{31}, M_{32}, M_{33}\} \\ C(M_{31}, O_3) &= 30 & C(M_{32}, O_3) &= 20 \\ C(M_{33}, O_3) &= 20 \\ G(1) &= \{Th_1, Th_2\} & G(2) &= \{Th_3\} \\ Gm(M_{31}, O_3) &= G(1) & Gm(M_{32}, O_3) &= G(1) \\ Gm(M_{33}, O_3) &= G(2) \\ Mr(M_{31}, O_3) &= \{(Cr_1, 10)\} & Mr(M_{32}, O_3) &= \{(Cr_1, 10)\} \\ Mr(M_{33}, O_3) &= \{(Cr_1, 9)\} \end{aligned}$$

図 5: リアルタイムオブジェクト O_3

$Sm(o)$	オブジェクト o のメソッド集合
$C(m, o)$	オブジェクト o の持つメソッド m の実行時間
$Ms(m, o)$	オブジェクト o の持つメソッド m が呼び出す他のオブジェクトのメソッドのマルチ集合
$G(i)$	スレッドグループ i のスレッド集合
$Gm(m, o)$	メソッド m を実行するスレッドグループ
$Mr(m, o)$	クリティカルリージョンとその実行時間のマルチ集合
where $\forall i, j, i \neq j, G(i) \cap G(j) = \emptyset$.	

オブジェクト O_3 を本モデルによってモデル化する。図 5 に示すように、スレッド T_1 と T_2 はメソッド M_{31} と M_{32} の実行を担当し T_3 は M_{33} の実行を担当する。オブジェクト内には一つのクリティカルリージョンがあるものと仮定する。 M_{31} の実行中は、クリティカルリージョンを 10 msec 実行する。 M_{32} の実行中はクリティカルリージョンを 9 msec 実行する。 O_3 の全てのメソッドの実行時間は以前と同じとする。

本例題のスケジューラビリティを調べる。先の例題の O_3 を変更しただけで、他は全て同じタスクセットである。

したがって、 P_1 と P_2 のブロッキング時間について考えれば良い。

O_3 を変更しても P_2 による P_1 のブロッキング時間は同じである。なぜならば、 O_3 は O_2 の M_2 を呼びだし、これが P_2 によりブロックしている場合があるからである。したがって、 P_2 による P_1 のブロッキング時間は以前と同じ 30 msec となる。

P_3 による P_1 のブロッキング時間は 9 msec となる。メソッド M_{32} はクリティカルリージョン実行時、 M_{31} または M_{32} の実行をブロックする可能性がある。この時間が 9 msec である。したがって、 P_1 のブロッキング時間は P_2 がブロックする 30 msec と P_3 がブロックする 9 msec の和で 39 msec となる。同様に P_2 のブロッキング時間は 9 msec となる。

表 2 は解析結果である。この表を使つてのタスクセットのスケジューリング可能性計算結果を以下に示す。このようにタスクセットはスケジューリング可能になったことが分かる。

$$\begin{aligned} & \frac{C(1)}{T(1)} + \frac{C(2)}{T(2)} + \frac{C(3)}{T(3)} + \max\left(\frac{B(1)}{T(1)}, \frac{B(2)}{T(2)}\right) \\ &= 0.4 + 0.1 + 0.075 + \max(0.195, 0.03) \\ &= 0.770 < 3(2^{1/3} - 1) = 0.780 \end{aligned}$$

2.3 タイム・フェンス機構

静的な面からリアルタイムオブジェクトモデルを見てきた。リアルタイムオブジェクトモデルは実行時に時間制約が満たされているかどうかを調べる機構として、タイム・フェンス機構 [6] を導入している。

タイム・フェンスは実時間タスクに関係づけられている。フェンスはオブジェクトの時間制約を実行時に検査するためのものである。もし、あるオブジェクトが他のオブジェクトを呼び出す時、呼び出されるオブジェクトのメソッドの実行時間および交信にかかる時間と遅れから、呼びだし側オブジェクトのタイム・フェンスを超えないかどうか検査する。もしフェンスを超えないようだったら、実際にオブジェクトを呼びだし、そうでなければ時間制約エラー処理を行なうという機能である。

3 RTC++ の概要

スレッド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。図 6 は RTC++ におけるアクティブオブジェクトのクラス定義例である。

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付加されている点である。**activity** 部でオブジェクトのスレッドを定義している。**activity** 部を省略した場合は単一スレッドとなる。

図 6 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** メソッドが定義されている。

```

1 active class Example1 {
2 private:
3   char   buf[BUF_SIZE];
4   int    count;
5   int    background();
6 public:
7   int    read(char* data, int size)
8         when(count > size);
9   int    write(out char* data);
10  int    open();
11  int    close();
12  Stat   statistic();
13 activity:
14  slave   read(char*, int);
15  slave   write(out char*);
16  slave   open(), close();
17  slave[5] statistic();
18  master  background()
19         cycle(;;0t30m;;);
20 };

```

図 6: アクティブオブジェクト宣言例

3.0.1 activity 部

activity 部においては、マスター / スレーブ スレッドを定義することができる。マスター・スレッドはメッセージ受信に関係なく独立に実行されるスレッドである。マスター・スレッド宣言ではスレッドが処理する関数名を宣言する。18 行目の記述はマスター・スレッドの使用例である。ここでは 30 分ごとに **background** ルーチンが呼ばれるようなバックグラウンド処理のためのスレッドを宣言している。**cycle** 文は周期的タスクを記述するための構文である。詳しくは 3.2.1 節で説明する。0t30m は 30 分を意味する。時間定数の例を以下に示す。“0t8h20m30s10.10” は、8 時 20 分 30 秒 10 ミリ秒 10 マイクロ秒を表現している。これは相対時間を意味する。絶対時間を表現するには何年何月何日から記述する必要がある。“0t1990Y05M15D12h15m20s” は、1990 年 5 月 15 日 12 時 15 分 20 秒を意味する。

スレーブ・スレッドは他オブジェクトからのメッセージを受信し処理するためのスレッドである。スレーブ・スレッド宣言時にそのスレッドが処理すべきメソッド名のリストを宣言する。14, 15 行目では、それぞれ、一つのスレッドが **read**, **write** メッセージを処理することを宣言している。16 行目では、一つのスレッドが **open**, **close** メッセージを処理することを宣言している。

複数のスレーブ・スレッドが同一の複数のメソッドのための処理を行ってもよい。この時、それらのスレッドをスレッド・グループと呼ぶ。17 行目はスレッド・グループの宣言例である。最大 5 つのスレッドが **statistics** メッセージを処理することを宣言している。スレッド・グループはブライオリティ・インバージョンの発生を押えることに貢献する。

なお、**activity** 部で宣言されていないメソッド全てに対しては一つのスレーブ・スレッドが割り当てられる。オブジェクト内に複数スレッドが存在する場合は、**region** 文を使って内部変数の排他制御をする必要がある。**region** 文は 3.3 節を参照のこと。

表 2: 図 4+5の時間情報 (単位 msec)

Process	Period(T)	Deadline	Execution(C)	C/T	Blocking(B)	B/T
1	200	200	80	0.4	39	0.195
2	300	300	30	0.1	9	0.03
3	400	400	30	0.075	0	0

```

1 Example1 *v;
2 v = new Example1 priority 4;
3 v = new (LOCAL) Example1 priority 4;
4 v = new (host1) Example1 priority 4;

```

図 7: アクティブオブジェクト生成例

3.0.2 ガード文

各メソッド宣言時には以下のようにガード文が記述できる。以下の例では“count > size”が真である時に限り、read関数を実行される。when文内で参照可能な変数はメッセージのパラメータとインスタンス変数である。

```

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

```

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。onwait文内で宣言される関数はオブジェクトから参照可能な関数である必要がある。

```

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

```

3.0.3 オブジェクト生成

図7はアクティブオブジェクトの生成例である。2行目では、実行しているプログラムと同じ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。3行目では、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。4行目では、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。なお、host1はRTC++が提供しているHostクラスのインスタンスでなければならない。

3.1 アクティブオブジェクト間通信とパラメータの受渡し

アクティブオブジェクト間通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。メソッドのアーギュメント定義部にin, inout, outという属性を付けることにより、データの授受関係を明示するようにしている。分散環境でのデータの授受では、Argus[2]同様ユーザにデータのマーシャリング、アンマーシャリングのメソッドの記述を強いる方式を採用している。アクティブオブジェクトのメッセージアーギュメントに現れるオブジェクトのクラスはpack, unpackメソッドが定義されてなければならない。packメソッドはマーシャリング時に、unpackメソッドはアンマーシャリング

```

1 class Example2 {
2 private:
3   int i;
4   char *sp;
5 public:
6   pack();
7   unpack();
8 };
9 Example2::pack()
10 {
11   pack i;
12   pack sp;
13 }
14 Example2::unpack()
15 {
16   i = unpack int;
17   sp = unpack char*;
18 }
19 active class Example3 {
20 public:
21   int func1(int a1, Example2 *a2);
22   int func2(int);
23 activity:
24   slave func1(int, Example2),
25         func2(int);
26 };
27 void
28 afunc()
29 {
30   Example2 *a = new Example2;
31   int i, j;
32   Example3 *ep;
33   ep = new ( host1 ) Example3;
34   j = ep->func1(i, a);
35 }

```

図 8: アクティブオブジェクトとの通信

時にそれぞれ呼び出される。なお、intやcharのような基本データクラスは暗黙のうちにこれらメソッドが定義されている。これらを使った例題を図8に示す。

3.2 時間制約プログラミング支援

時間制約はオブジェクトのメソッド単位とステートメント単位に宣言することが可能である。

3.2.1 メソッド単位の時間制約文

図10はリアルタイムオブジェクトの記述例である。readメソッドは20msec以内に終わらなければならないと定義されている。もし20msec以内に終わらなければread_abort関数が呼ばれる。within文は実際に経過した時間による時間制約である。すなわち、CPU消費時間に関係なく20msecを経過してもreadメソッドが終了しない場合、read_abort関数が呼ばれる。

bound文によって、最悪CPU消費時間を宣言することもできる。図10のwithinをboundキーワードに変更したと仮定する。この場合、もしCPUを20msec使用しても処

```
cycle(<starttime>; <endtime>; <period>; <phase>; <deadline>;);
```

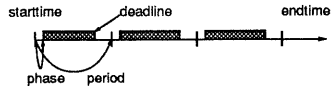


図 9: Cycle 文

```
1 active class Example1 {
2 private:
3   int  read_abort();
4   int  write_abort();
5   // ...
6 public:
7   int  read(char* data, int size)
8       when(count > size)
9       within(0t20)
10        timeout(read_abort());
11   int  write(out char* data)
12       within(0t20)
13       timeout(write_abort());
14   // ...
15 }
```

図 10: 時間制約を含んだ例

理が終了しなければ `timeout` 文で宣言されているアボート関数が呼ばれるという意味になる。

図 6 の 18 行目の記述で現われている `cycle` 文は周期的タスクを記述するための構文である。一般に図 9 の上部に示すような記述をする。ここで、`< >` で囲まれる文字列はメタ記号とする。`<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、`<phase>` で示される時間からマスタースレッドの実行が始まり `<deadline>` で示される時間以内に終了する必要がある。これらの関係を図 9 の下部に示す。

`<starttime>` 省略時は、オブジェクトが生成された時が開始時間となる。`<endtime>` 省略時は、無限に続く周期タスクとなる。`<period>` を省略することは出来ない。`<phase>` 省略時は `0t0` と同じである。`<deadline>` 省略時は次の周期の始まりがデッドラインとなる。

3.2.2 ステートメント単位の時間制約文

ここで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について説明する。以下のプログラムはステートメントレベルでの経過時間制約文である `within` 文の例である。ここで `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。

```
1 within(time) {
2   // <within body>
3 } except {
4   case timeout:
5   // <timeout body>
6 }
```

`within` 文以外に `before`、`after` 文がある。`before` 文は終了時間を規定し `after` 文は実行の開始時間を規定する。ステートメントレベルでの CPU 消費時間の最悪値を宣言する `bound` 文も提供されている。また、`cycle` 文は

プログラム本体中でも使うことができる。これらのシンタックスは `within` 文と同様である。

3.3 クリティカルリージョン

アクティブオブジェクト内の複数スレッドが相互排除しなければならない領域(クリティカル・リージョン)を実現するために、RTC++ では以下のように `region` 文を提供している。なお、クリティカルリージョンの実行では優先順位継承が行なわれる。

```
1 region (rr) {
2   // <region body>
3 } except {
4   case abort:
5   // aborted
6 }
```

3.4 保護領域

プログラムが任意の時点で他のプログラムから強制終了させられたり、時間切れで強制終了させられるのを保護するために保護領域を設定することができる。以下のプログラムにおいて `shield` キーワード以降のブロック文は保護された領域である。

```
1 shield {
2   // <shield region>
3 }
```

4 今後の課題

4.1 スケジューリング可能性解析ツール

RTC++ 言語の特徴はプログラミング言語において時間制約が記述されている点である。従来の実時間記述ツールでは時間制約はツール上の言語で記述し、実装言語では、時間制約記述が欠落するという欠点があった。RTC++ 言語で記述したプログラムから 2 章で述べた方法により、RM スケジューリングでのスケジューリング可能性解析が可能である。この時、時間制約を静的に決めるために動的言語機能を制限する必要がある。

また文献 [7] において、スケジューリング可能性解析システムである Scheduler123 というシステムを報告している。Scheduler123 は時間制約を持つタスクセットが様々なスケジューリングアルゴリズムでスケジューリング可能かどうかを調べることができる。RTC++ のソースプログラムから時間制約だけを取りだし Scheduler123 の入力として与えるようなシステムを現在設計している。

4.2 言語の拡張

オブジェクト間交信は同期型交信のみしか提供していない。しかし、ユーザはアプリケーションプログラムを最

適に実現するために様々な交信機能を要求するであろう。例えば、センサーオブジェクトがコントロールオブジェクトに定期的に情報を送っているようなシステムを考える。コントロールオブジェクトの処理が多くなってセンサーオブジェクトから情報を処理し切れなくなった場合、輻輳制御を必要とする。これは従来オペレーティングシステムが適当に制御するかオペレーティングシステムプリミティブとしてユーザに解放されている。プログラミング言語として分散システム上での交信プリミティブを提供する場合、輻輳制御を言語機能として要求するユーザもいるであろう。

そこで、言語レベルから交信プロトコルを記述できるような枠組を提供しようと考えている。

例えば、図 11 では、`func1` 呼び出しの通信プロトコルとして `timefence`、`func2` 呼び出しの通信プロトコルとして `idenpotent` を、`func3` 呼び出しのプロトコルとしてユーザ定義プロトコル `userproto` を定義している。`idenpotent` や `timefence` は `RTC++` が提供する基本プロトコルとする。ユーザ定義プロトコルは `protocol` 記述子により記述可能とする。

`protocol` 記述子は `class` 記述子と同様構造体の定義およびメンバ関数を定義できる。図 11 において、10 行目は `userproto` の送信側プロトコルを 14 行目は受信側プロトコル記述定義である。プロトコル定義部では、交信のための低レベルカーネルプリミティブ（例えば `UDP`）を使って実現することになる。また、送信/受信側オブジェクトでプロトコルハンドラに対して挙動を変更できるように `ioctl` 機構も定義できる。

送信側オブジェクトは例えば以下のようにして `ioctl` を発行しプロトコルハンドラを制御する。

```
class Example4 *ep;
```

```
ep->func1.ioctl(PESMISTIC_CNTL);
```

```
ep->func3.ioctl(URGENT);
```

受信側オブジェクトは以下のようにしてプロトコルハンドラを制御する。

```
Example4::func3(int a)
{
    ...
    func3.ioctl(QUENCH);
    ...
}
```

5 おわりに

本稿では、まず、リアルタイムオブジェクトモデルを紹介した。次に、本モデルを基に設計した言語 `RTC++` についての概要を紹介した。そして、今後の課題について述べた。`RTC++` は `Sun3`、`NEWS1700` 上の `ARTS` カーネルで稼働している。

`RTC++` の詳細な言語仕様および実現方法、評価については [9] を参照されたし。

参考文献

[1] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer: Object-oriented real-time language design:

```
1 protocol userproto { ... };
2 active class Example4 {
3 public:
4   timefence int func1(int, char);
5   idenpotent int func2(int);
6   userproto int func3(int);
7 activity:
8   slave func1(int,char), func2(int), func3(int);
9 };
10 protocol userproto::sender()
11 {
12   /* 送信側プロトコル定義 */
13 }
14 protocol userproto::receiver()
15 {
16   /* 受信側プロトコル定義 */
17 }
18 protocol userproto::ioctl(cmd)
19 {
20   /* プロトコル制御 */
21 }
```

図 11: プロトコル記述

Constructs for timing constraints, *Proceedings of OOPSLA-90*, pp. 289–298, October 1990.

- [2] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl: Argus reference manual, Technical Report Memo 54, MIT Programming Methodology Group, March 1987.
- [3] C.L. Liu and J.W. Layland: Scheduling algorithms for multiprogramming in a hard real time environment, *Journal of ACM*, Vol. 20, No. 1 (1973), pp. 46–61.
- [4] Clifford W. Mercer and Hideyuki Tokuda: The arts real-time object model, *Proceedings of 11th IEEE Real-Time Systems Symposium*, pp. 2–10, December 1990.
- [5] L. Sha, R. Rajkumar, and J.P. Lehoczky: Priority inheritance protocols: An approach to real-time synchronization, Technical Report CMU-CS-87-181, Carnegie Mellon University, November 1987.
- [6] H. Tokuda and C.W. Mercer: Arts: A distributed real-time kernel, *Operating Systems Review*, Vol. 23, No. 3 (1989), pp. 29–53.
- [7] Hideyuki Tokuda and Makoto Kotera: A real-time tool set for the arts kernel, *Proceedings of the 9th IEEE Real-Time Systems Symposium*, December 1988.
- [8] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda: Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, A. Yonezawa and M. Tokoro (eds.), *Object-Oriented Concurrent Programming*, pp. 55–89, MIT Press, 1987.
- [9] 石川 裕, 徳田 英幸: 分散型実時間プログラミング言語 `rtc++`, コンピュータソフトウェア, Vol. 9, No. 2 (1992).