ロボット制御プログラムの無停止更新手法

堀 内 英 一*

Nonstop Update of Running Robot Controller Programs

Eiichi Horiuchi*

This paper presents a method to update running RTLinux robot controller programs (RTU) on the fly. The proposed method exploits thread swapping. This paper focuses on the safety conditions of thread swapping, platform requirements, realtime issues, and uncertainties in RTU completion time. This paper introduces RTU-able software design by layered modules which enable the update of servo functions and robot model global variables. We have developed a RTU-able mobile manipulator controller, evaluated its performance by worst case results, and shown its practical applicability.

Key Words: Nonstop Update, Thread Swapping, Layered Modules, RTLinux, Agile System.

1. は じ め に

近年製造業に対する顧客の要求が多品種少量化する一方、海外との競争のため徹底したムダの排除が要求されている。生産方式も生産効率重視の従来のラインから、仕掛品や在庫がなく売れただけ生産する少量生産に適するセル生産方式へと変わりつつある。加えて変化するニーズに答えられる“アジャイル”な生産方式も、競争に残る条件の一つとして認識されるに至った。この状況は 1972 年にすでに予測 [1] されていたが、製造分野向けのロボットにも十数年前の状況 [2] [3] と比べて、一層厳しい条件が要求されている。

製造現場でのロボット利用上の障害の一つが、作業の切替や変更に伴う段取替えの時間ロスである。セル生産方式にロボットを導入する場合、多能工的タスクが要求されるのでロボットの作業切替の頻発は避けられない。メーカー努力でハードウェア起因の段取時間の短縮は進んでいるが、プログラム変更やメンテナンスなどソフトウェア起因の段取時間の短縮は進んでいない。状況変化を事前に想定した場合分けで対応するか、できなければロボットを停止してプログラム変更するのが現状である。ロボット停止は問題だが、これ以外に問題点が二つある。一つはすべての状況を事前に想定することは不可能という問題である。もう一つは仮に想定できても、コーディングしたプログラムの大部分が実際に使われないというムダである。

インターネットが普及した今日、ミッションクリティカルなオンラインサービスを提供するサーバが多数稼動する。この種の

システムではサービスの常時可用性を考慮して、メンテナンス時などにサービスを無停止のままプログラム更新ができる技術が実用化されている。もし同じことがロボットで可能ならば、ロボットの無停止稼動に加えて、必要なときに必要なだけコードをムダなく追加・更新できるアジャイルなシステムが実現できる。サーバなど純粋ソフトウェアではなくロボットなどハードウェア制御を含むシステムを対象として、実行中のユーザプログラムを無停止更新するプラットフォームに依存しない一般の枠組み (NSU) [4]、プラットフォームを特化した枠組み (RTU) [5] が提案されている。本論文では後者について、ロボット制御に必要なリアルタイム性に関する問題を考察し、より広いアプリケーションを目指したソフトウェア設計の方法論について述べる。また実際のロボット制御プログラムを用いて RTU の時間データを計測する。

無人化工場で長時間稼動するロボットアームは本手法の適用可能対象だが、近年長時間無停止稼動 [8] [9] が研究されている移動ロボットにもニーズはある。物理的なバッテリー交換を必要とするこれまでの移動ロボットは停止が避けられなかった。しかしバッテリー充電の自律的実行 [10]~[12] はその解決策の一つであり、自己充電可能なベットロボット [13] も市販されている。また技術的進歩の著しい燃料電池を搭載した移動ロボットも発表されている。ロボットは充電時には自由に移動できないが、そのシステムは通常の機能を維持できる。一般に稼働時間が伸びるほど、ロボットのソフトウェア更新が必要となる確率は増加する。充電と更新のタイミングが常に一致するわけではなく、必要時にすぐ更新できるのが望ましい。

原稿受付 2003 年 12 月 12 日

*産業技術総合研究所

*National Institute of Advanced Industrial Science and Tech.

2. スレッドスワッピング

2.1 スレッド

プログラム実行中の制御の流れは、一般にスレッドと呼ばれる。例えばC言語のコードでは、スレッド中で main 関数に始まり次々と関数が呼ばれる。プログラムはコード全体からアクセスできる外部変数を含むとともに、動的にメモリの確保・解放を実行し、これらの値は時間変化する。仮想的に時間を停止し、関数呼び出しのネスト、各関数の引数とローカル変数と実行位置、外部変数値、動的メモリの内容等の情報を考えると、これらはスレッド実行状態を一意に定義するパラメータとなる。本論文ではこれをコンテキストと呼ぶ。

ユーザ入力を待ちながらバックグラウンドで計算するプログラムは複数のスレッドを所有し、これはマルチスレッドと呼ばれる。いまマルチスレッドプログラムが実行中のスレッドを停止してその実行状態を取得し、並列的に別スレッドを起動してその実行状態を復元できたとする。外からは処理が続くように見えるが、スレッドは置き換わっている。これがスレッドスワッピングである。一般にスレッドはスレッド開始関数を呼び出して起動するが、一定の整合性を満たす限り、スワッピング時にまったく別の開始関数を呼び出してもよい。もし新たに追加したモジュールにある開始関数を呼び出せば、プログラムのコードを更新したことになり、これが無停止更新の基本となる。

Fig. 1 に無停止更新と通常の更新方法の違いを示す。無停止更新はコード全体ではなく更新モジュールだけをロードし、プロセスではなくスレッドを削除／生成するのでオーバーヘッドが小さい。ハードウェアを含む制御プログラムでは終了／初期化時間が無視できないが、無停止更新はプロセスを停止しないのでこれらの処理は省略できる。コンテキスト取得／復元の時間はコンテキストサイズに依存するが、サイズはスワップ時のスレッド実行位置に依存する。例えば関数呼び出しの深いネストの位置でスワップすると、コンテキストサイズが増大するだけでなくコンテキスト取得／復元処理自体が重くなる。時間的不確かさは実時間処理では問題だが、一つの解決策を次節で示す。

2.2 整合性

スワッピング時に満たすべき整合性には、構文的整合性と意味論的整合性がある。プログラム実行時の新モジュール読み込みは一般に実行時リンクと呼ばれ、新モジュールが参照するシンボル（関数や外部変数）は、そのモジュールまたはすでにリンク済みのコードで定義されている必要がある。これが構文的整合性であり、未定義シンボルはリンクで容易に検出できる。また関数を初期設定や行列計算に、外部変数を状態変数や制御変数に利用するのはプログラムの意味論である。例えば他のコー

Ordinary Update	Nonstop Update
	Context Acquisition
Termination of Processes	Termination of Threads
Reload of Entire Code	Reload of Updated Modules Only
Creation of Processes	Creation of Threads
Initializations	skip
	Context Restoration

Fig. 1 Ordinary update and nonstop update

ドで状態変数と解釈する変数を新モジュールで制御変数と解釈したり、古いコードが初期設定実行中に更新され新モジュールが行列計算の途中から実行再開したりすると、プログラムは意図どおりには動作しない。これが意味論的整合性であるが、その保証はプログラマの責任となる。

プログラマの負担を減じるには、更新に制限を加えることが必要である。コード変更の制限は有効だが無停止更新の利点が損なわれる。もう一つの方法は更新タイミングの制限である。本論文はスワッピングを実行するコード上の位置を交代点と呼ぶが、交代点は必ずしもコードの任意の位置に設定できる必要はない。交代点の位置を制限すれば、プログラマはその位置の整合性保証に集中できる。交代点設定の基準の一つはスレッドが交代点を通る頻度である。頻度が大きいほど更新開始イベント発生から更新実行までの時間遅れは小さい。交代点設定のもう一つの基準はコンテキストサイズである。コンテキストサイズが交代点に依存することは前述したが、サイズができる限り小さくなる位置に交代点を設定すれば無停止更新の時間を短縮できる。交代点ではさらにスワッピング実行の条件を設定できる。例えば不安定系ではスワッピングによる制御の遅れが致命的になる可能性があるが、制御遅れがあっても安全な場合に限り更新実行すれば問題は回避できる。

2.3 プラットフォームの条件

スレッドスワッピングを実装するための一般的要件は

- (1) スレッドの管理
 - (2) 新旧モジュールのバージョン管理
 - (3) 実行状態の取得／復元手続き
- の3点である。これを各種プラットフォームに適用する場合、プラットフォーム独自の条件が加わる。プラットフォームが満たすべき条件は
- (4) マルチスレッドのサポート
 - (5) 実行可能モジュールの実行時リンク
- の2点である。プログラム実行中に別モジュールを読み込める条件が(5)である。ロボットコントローラには一般にリアルタイム性が要求される。RTOSの一つでありオープンソースのフリーウェア[7]でもあるRTLinuxはロボット制御に適している。例えばRTLinuxは(4)をUNIX標準のPOSIXスレッドとしてサポートし、(5)をLinuxデバイスドライバに似た方式でinsmod/rmmodコマンドでサポートしている。

3. リアルタイムアップデート

3.1 RTU 可能コードへの変換

RTLinux [6] はLinuxカーネル外部に実時間カーネルを実装することで、Linuxの機能と実時間タスクを両立させたRTOSの一つである。RTU (リアルタイムアップデート) はRTLinuxをプラットフォームとして、実時間タスクに無停止更新機能を実装した、Cを開発言語とするソフトウェア設計の方法論である。通常の実時間プログラムをRTU可能コードに変換するために、RTUは先の条件(1)、(2)のためのユーザ向けマクロを用意するが、条件(3)はアプリケーション依存のためユーザが実装する必要がある。変換に当たり、ソースコードが次の条件を満たすスタイルで記述される必要がある。

Fig. 2 Dependency among layered modules

- (a) ソースコード上で交代点を明示的に指定
 (b) 安定モジュール／更新可能モジュールへの階層化
 (c) 更新可能部分の更新頻度に応じた階層化

ここで階層化とは、Linux デバイスドライバ[15]のそれと同じものを表す。条件 (a) は前述のとおり。条件 (b), (c) は更新範囲を小さくして更新処理を短縮するための条件である。整合性で述べたが、モジュール間には依存関係がある。例えばモジュール B の関数がモジュール A の関数や外部変数を参照する場合、B より先に A が実行時リンクされる必要がある。この制約は、モジュール A を更新するのに更新の有無によらず A を参照する全モジュールを再度実行時リンクする必要があることを意味する。Fig. 2 に示すように B が A、C が B に依存する場合、更新不要な B、C も再リンクが必要である。プログラムのシンボル（関数および外部変数）で更新可能性がないものを安定モジュールに組み入れ、ほかを更新頻度および関連性で階層化モジュールに分割することが、この問題の軽減に必要である。

3.2 更新スケジュールと時間的不確かさ

実行中の旧モジュールと交代すべき新モジュールが手元ですであると仮定する。RTU の更新手順は以下のとおりであり、更新回数に制限はない。

- (1) insmod で新モジュールをリンク、新スレッドは待ち状態
- (2) 実行中の旧スレッドの終了
- (3) 旧スレッドのコンテキストを取得
- (4) 待ち状態の新スレッドの実行再開
- (5) 新スレッドでコンテキストを復元
- (6) rmmod で旧モジュールを削除

ここでリアルタイムタスクが停止される時間が (2)～(5) に当たる。RTU の時間的不確かさは次の 3 点が考えられる。

- (A) (3), (5) の処理時間がコンテキストサイズに依存
- (B) (2), (4) の処理時間がプラットフォームに依存
- (C) (3) と (4) の間はリアルタイムタスクが皆無

ここで (C) は処理が一時的に Linux に移り、割り込みの影響を受けることを意味する。RTLinux では (B) は安定しており無視できる。このスケジュールは新旧スレッド交代の時間順序を規定しており、プラットフォームが SMP かどうかに依存しない。

3.3 リアルタイムタスク

周期的リアルタイムタスクを考える。デッドラインまでに必ず完了すべきタスクがある場合、そのタスクはハードリアルタイム [16] といわれる。保証型スケジューリングはデッドライン遵守を保証するもので、最悪の場合のタスク完了時間が予測できることを前提とする。しかし実システムの多くは時間的不確定性をもち、最悪の場合の定義が困難なことが多い。また保証型は悲観的予測を行うので、実際利用する以上の計算機リソ

Fig. 3 Hard realtime tasks

スを要求したり、性能仕様を低めに設定しがちななどの問題がある。デッドラインが努力目標の場合はベストエフォート (BE) 型スケジューリングが有効である。これは時間遵守を性能仕様の一部とみなし、超過のリスクを全体的コストで考えるものである。Fig. 3 に両者の違いを示す。タスク A、B が予測最悪値により時間遵守でスケジュールされているときに新タスク Cが入った場合、保証型は時間超過を防ぐため C を拒絶するが、BE 型は全体的コストを考慮して C を受け付ける。タスクが予測値以下の時間で終了した場合、BE 型では結果的に時間遵守できる場合があるが、保証型では生じた時間的余裕は使われない。

RTU は時間的不確かさをもつので、保証型ではなく BE 型のスケジューリングを行う。BE 型は 2 足歩行など不安定系の制御に不適にみえるが、制御変数を監視して一定回数変化がないことを条件に更新実行する、多少の制御遅れがあっても安全な歩行周期の特定の点を交代点に設定するなど、対応は可能である。ロボット制御で基準とすべき時間はサーボ周期なので、サーボ周期に対する更新所要時間のパーセンテージが RTU の評価指標として妥当である。交代点の設定基準など更新所要時間を短縮する手法についてはすでに述べた。更新開始イベントは特に限定せず、タイマ、ユーザプログラムからのメッセージ、サーボループ回数、割り込み信号などを利用する。

4. 更新タスク

4.1 階層化モジュール

プログラム更新には、モジュールの追加、置換、削除、復元の四つの場合がある。復元とは、保存していた旧モジュールとコンテキストを用いてプログラムを元に戻すエラーリカバリである。RTU はモジュールを更新単位として、関数、外部変数などリンカ対象のシンボルレベルで、前述の整合性を満たす範囲で自由に更新できる。以下ではロボット制御に特化した更新タスクを考える。本手法は Fig. 4 に示す階層化モジュールを使用する。プログラム全体は、アプリケーション層、ライブラリ層、非更新の安定層に階層化される。アプリケーション層は制御計算などのためにライブラリ層の関数を呼び出す。ライブラリ層の関数はハードウェアや通信などのために安定層の関数を呼び出す。最下位層の安定層は、ハードウェア関係の物理定数やアドレス、通信関連の関数、行列計算関数などを管理する。ただし階層化は相対的なものであり、タスクに応じた構成が重要である。前述の依存関係より、モジュールは常に自分または下位階層モジュールの関数と外部変数を参照するので、被更新モジュール

Application Layer	
Module Local Variables	Application Functions Thread Function
Library Layer	
SharedApplicationVariables	Application Library Functions
Application Context Buffer	Application Context Functions
Module Local Variables	
Stable Layer	
Shared System Variables	System Library Functions
System Context Buffer	System Context Functions
Module Local Variables	
Global Variables	Functions

Fig. 4 Layered modules for robotic applications

Table 1 Thread-related RTLinux APIs

pthread_create	create realtime thread
pthread_make_periodic_np	evoke realtime thread periodically
pthread_wait_np	suspend thread until next period
pthread_wakeup_np	wake up realtime thread
pthread_suspend_np	suspend current thread
pthread_delete_np	delete realtime thread

Fig. 5 Layered modules for servo computation update

ルのコンテキスト取得／保存／復元は一階層下のモジュールが管理する。更新が複数階層にわたる場合は、更新しない層で最上位の層が管理する。実時間スレッド開始関数は、最上位層のアプリケーション層が管理することとする。次に具体的な更新タスクを述べる。

4.2 サーボ制御計算の変更

計算部分に不具合がありパラメータではなく制御計算を変更したい状況はよく生じる。ここではサーボ制御計算を変更する更新タスクを述べる。Table 1 にスレッド関連の RTLinux の主な API を示す。RTLinux で周期実行の無限ループを実装するには、pthread_wait_np 関数を用いて以下のように行う。

- (1) 無限ループ内部に pthread_wait_np 関数を置く
- (2) pthread_make_periodic_np 関数で実行周期を指定
- (3) 無限ループは開始後に (1) の位置で一時停止
- (4) その後は (2) の指定周期で実行再開／一時停止

コード更新はアプリケーション層モジュールの置換 (Fig. 5) で行う。これは実時間スレッド開始関数 (start_routine) 内部で

無限ループを周期実行し、無限ループが呼び出す制御計算関数 (servo_comp) を同モジュールに含めることで実現できる。交代点は無限ループ内部に 1 箇所だけ設定する。これは、サーボ周期で必ずスレッドが通過する、最初に呼ばれるスレッド開始関数に設定すればコンテキストサイズが小さくなる、プログラマの整合性保証の負担が軽くできるという理由による。この場合のコンテキストは、start_routine 関数の引数とローカル変数と動的割当メモリ、アプリケーションモジュールが定義する外部変数からなる。交代点は servo_comp 関数外部にあるので、コンテキストはこの関数の実行状態は含まない。

例としてデジタル PID 制御の変更を説明する。現時刻誤差など各ループで再計算する情報は servo_comp 関数のローカル変数で定義し、誤差履歴など蓄積する情報はモジュールの外部変数で定義する。例えば servo_comp 関数が実装する微分動作や積分動作は、モジュールを置換して新しい servo_comp 関数を呼べば変更できる。変更で生じる操作量 u の差が問題なら、バンプレス切り替えを行う。次のようなループ N 回分の移行期間を設ければよい。 $u = u_{new}(t_i)(1 - \alpha^i) + u_{old}(t_i)\alpha^i, (0 < \alpha < 1, i = 0, 1, 2, \dots, N - 1)$

4.3 ロボットモデルの変更

アームの第 L 関節から先／移動体の第 M 脚が何かの原因で失われた、アーム第 N 関節が自由関節になったなど、ロボットの運動自由度が突然変化した場合を考える。まずシステムが無傷、ロボットの物理的状態と自由度と環境の状態が完全に検知可能、ロボットが物理的に動けることを仮定する。次に制御プログラムが失敗回復可能状態、すなわち制御の実行スレッドが生きていて交代点を定期的に通過する状態にあると仮定する。この場合に、ユーザから何かの更新開始イベントをロボットに送信できれば、スレッドが交代点に到達した時点で RTU が起動できる。

状況把握に時間を要するため短時間での更新は無理だが、システム停止の必要がないことが RTU の利点である。更新項目は、変化後のロボットの自由度に対応した運動学／動力学モデルである。基本的な考えは次の 2 点である。

- (1) 変化前後のモデルをオフラインでモジュール化
 - (2) オンラインで変化を同定してモジュールを選択
- 自由度変化に対応したモデルを共通部分と差分部分にオフラインでモジュール化すれば、変更は差分で対処できる。具体的な更新対象には、ヤコビ行列計算や動力学計算などの関数に加えて、状態変数、制御変数、ハードウェア関連の物理定数など外部変数も含まれる。アプリケーション層は前と同じで、これらのシンボルを安定層ではなく更新可能なライブラリ層に含めておけば、Fig. 6 のようにアプリケーション層とライブラリ層のモジュールの置換で更新できる。

5. 移動アームコントローラの無停止更新

5.1 システム

RTU を 7 自由度アーム (PA10) を搭載した移動アーム制御プログラムに適用した。使用したのは協調型パワーアシスト装置 [14] で実際に使われるコードである。装置の外観を Fig. 7 に示す。アーム先端にはハンドと力センサを装備する。システム

Fig. 6 Layered modules for robot model update

Fig. 7 A mobile manipulator

Fig. 8 System setup for mobile manipulator

構成を Fig. 8 に示す。Linux 側アプリケーションは更新開始イベントを実時間 FIFO を介して RTLinux 側スレッドに送信し、実時間スレッドは実行結果を返す。この事例ではサーボ制御計算関数を更新するタスクを設定して RTU の時間的評価を行う。実用を考慮してチャンピオンデータではなく最悪実測値で評価する。アーム部分のサーボ周期は 2 [ms] であり、これが評価の時間基準となる。

モジュール階層化は Fig. 5 が基本であるが、コード更新はモジュール追加で行った。Fig. 9 に実装した階層を示す。これはオリジナルコードの変更を最小にするためであり、被更新部分のコードは残るが更新後はアクセスされない。コード変更はバージョン管理、新旧スレッド管理、スワップ時初期化処理スキップの 3 点あり、668 行のソースファイルを、マクロ追加 5 箇所、マ

Fig. 9 Layered modules for mobile manipulator

クロへの置換 6 箇所、更新開始イベント受信用追加コード 4 行で変換できた。検証目的の更新タスクは、アームにゼロ速度指令を送るサーボ計算関数を、同じくゼロ速度指令を送る別の関数で置き換える作業とした。スレッド開始関数のコードを Fig. 10 に示す。ここで交代点に該当するのが switch 文であり、取得／復元されるコンテキストは double 型変数 1 個 (8byte) である。ユーザからの更新開始イベントは GET_TASK メッセージとして旧スレッドが受信し、コンテキスト取得後停止する。新スレッドは PUT_TASK メッセージを受けてコンテキストを復元して実行再開する。所要時間は旧スレッド clock_gettask と新スレッド clock_puttask の差で計測する。

コンテキストサイズが 8 [byte] の場合と意図的に 10 [kbyte] とした場合の所要時間を計測した。前者 97 回、後者 86 回の試行結果を Fig. 11 に示す。縦軸は時間の対数を表し、プラットフォームは RTLinux 3.1, Pentium III 1 GHz SMP である。新モジュール追加の成功と、新モジュールによる既存モジュールの正常な参照は、RTLinux の lsmod コマンドで確認できた。両者とも最悪値はほぼ等しく、サーボ周期 (2 [ms]) の 97% という結果を得た。これは、コンテキストサイズによらず、最悪でもループ 1 回の時間でプログラム更新できることを意味する。両者の平均値からは、前述の時間的不確かさ (A) を評価できる。また不確かさ (C) の影響が大きいことが確認できる。Linux のサービスをロボットが利用する最小限にするのが解決策である。

6. 結 論

本論文は、RTLinux をプラットフォームとするロボット制御プログラムを対象として、実行中のプログラムを無停止更新する手法 (RTU) を提案した。RTU の基礎はスレッドスワッピングにある。安全に更新できる整合性条件、プラットフォームの条件を明らかにし、実時間性の問題、時間的不確かさの問題を述べた。RTU をロボット制御に適用するためのモジュール階層化について、サーボ制御更新、ロボットモデル更新の際のノ


```

void *start_routine(void *t){
    double s_time;
    MonitorMessage msg;
    RTU_GOTO(t,loop)
    /* Initializations of ARCNET, matrices, sensors, ...*/
    s.time = gethrtime_dbl();
    while(1){
        loop:
        pthread_wait_np();
        if (rtf_get(FIFO,&msg,sizeof(msg)) == sizeof(msg)){
            switch (msg.command) {
                case START_TASK:
                    pthread_make_periodic_np(SELF,gethrtime(),PR);
                    start_arcnet();
                    break;
                case PUT_TASK:
                    restore_context(&s.time);
                    pthread_make_periodic_np(SELF,gethrtime(),PR);
                    clock_puttask = gethrtime();
                    break;
                case GET_TASK:
                    clock_gettask = gethrtime();
                    acquire_context(&s.time);
                    pthread_suspend_np(SELF);
                    break;
                case STOP_TASK:
                    pthread_suspend_np(SELF);
                    end_arcnet();
                    break;
            }
        }
        servo_comp();
        rtf_put(FIFO_DATA,(char*)&rd,sizeof(rd));
    }
    return 0;
}

```

Fig. 10 Thread function source code

Fig. 11 Elapsed time of RTU

ウハウを述べた。RTU を移動アームに適用し、チャンピオンデータではなく最悪値で所要時間を評価し、RTU が実用に足

ることを検証した。

謝辞 移動アームの制御プログラムを提供していただいた田窪朋仁氏（現大阪大学）に感謝します。

参考文献

- [1] M. Minsky: "Mini-robot Proposal to ARPA," AI Memo, no.251, MIT A. I. Laboratory, 1972.
- [2] 特集「FA とロボット」, 日本ロボット学会誌, vol.5, no.3, 1987.
- [3] 特集「CIM とロボット」, 日本ロボット学会誌, vol.10, no.3, 1992.
- [4] 堀内: "連続稼動ロボットのためのノンストップアップデート", 第8回ロボティクスシンポジウム予稿集, pp.457-462, 2003.
- [5] E. Horiuchi: "Nonstop Update of Mission Critical Robotic Services," In Proc. of IEEE/RSJ IROS 2003, 2003.
- [6] V. Yodaiken and M. Barabanov: "A Real-Time Linux," In Proc. of Linux Applications Development and Deployment Conf., Jan. 1997.
- [7] RTLinux: <http://www.fsmlabs.com/products/openrtlinux/> Accessed on 2004 年 4 月 2 日.
- [8] W. Burgard, A.B. Cremers, et al.: "The Interactive Museum Tourguide Robot," In Proc. of the 15th National Conference on Artificial Intelligence, Madison, WI, 1998.
- [9] S. Picault and A. Drogoul: "Robots as a Social Species: the MICRobES Project," In Proc. of SIA'2000, AAAI Fall Symposium Series, 2000.
- [10] S. Yuta and Y. Hada: "Long Term Activity of the Autonomous Robot—Proposal of a Bench-mark Problem for the Autonomy," In Proc. of the 1998 IEEE/RSJ IROS, pp.1871-1878, 1998.
- [11] S. Oh, A. Zelinsky and K. Taylor: "Autonomous Battery Recharging for Indoor Mobile Robots," In Proc. of Australian Conf. on Robotics and Automation, 2000.
- [12] M.C. Silverman, D. Nies, B. Jung and G.S. Sukhatme: "Staying Alive: A Docking Station for Autonomous Robot Recharging," In Proc. of the 2002 IEEE Int. Conf. on Robotics and Automation, pp.1050-1055, 2002.
- [13] M. Fujita and H. Kitano: "Development of An Autonomous Quadruped Robot for Robot Entertainment," Autonomous Robots, vol.5, no.1, pp.7-18, 1998.
- [14] 田窪: "非ホロノミック拘束を利用する物体の人間・ロボット協調搬送に関する研究", 筑波大学工学研究科, 2003.
- [15] A. Rubini and J. Corbet: Linux デバイスドライバ第2版. オライリー・ジャパン, 2002.
- [16] G.C. Buttazzo: Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications. Kluwer Academic Pub., 1997.

堀内英一 (Eiichi Horiuchi)

1959 年 10 月 2 日生。1984 年京都大学工学部精密工学研究科修士課程修了。同年旧工業技術院機械技術研究所。現在、産業技術総合研究所所属。移動ロボット、遺伝的アルゴリズム、スキルの研究に従事。計測自動制御学会会員。(日本ロボット学会正会員)