ロボットと計算機の界面におけるプログラミング

横　山　大　作[†1]

近年、ロボットのためのプログラミング環境が次第に整備され、広まり始めている。ロボット内部の複数の計算機、センサ、外部の高速計算機などを統合して利用するためには分散プログラミングを行う必要があり、簡単に利用するためには通信の隠蔽などの機構が欠かせない。本稿では、タプルスペースによるルールベースの記述が可能なプログラミング環境 “coalsack” を紹介し、その設計とプログラム事例を紹介する。また、ロボットのためのプログラミング環境をいくつか紹介し、求められる性質と今後の方向性について考察する。

1. はじめに

一般的に、ロボットという言葉は独立したひとかたまりの機械構造物とイメージされることが多い。電源容量や重量の関係から、小規模な計算機しか搭載できないことも多く、ロボットプログラミングはそのような小型計算機上で組み込み的環境を意識したプログラミングを行ってきた。しかし、近年のロボットにおいては、ロボット内部の単体計算機で処理が完結することなく、ネットワークで接続された外部の記憶や計算機資源を利用して、人間に対する高度なサービス提供を実現していることがしばしばである。また、ロボット内部にも多数の計算機が導入され、体内ネットワークを構築し、複数の計算機を協調的に利用することで制御を実現していることが多くなってきた。これらの計算機は違う OS、違うプログラミング言語で動作していることも多い。さらに、施設内や街中に設置されている、ネットワーク化された多数のセンサ群からのデータを収集統合し、高度な判断に用いることも可能になりつつある。

このように、ロボットのプログラミングにおいては、ロボットの内部の小型計算機と外部の大型計算機群やセンサ群との間の界面、ロボット内部に多数存在する計算機間の界面など、多くのレベルの界面が存在し、それらを簡単に取り扱えるような分散プログラミング環境が要求されることになる。

また、冷却機構などの制約により単体計算機性能の向上は近年頭打ちにあり、メニーコアと呼ばれる多数の計算機コアを持つ CPU を利用したり、安価なコモディティ部品によって構成された PC クラスタを用いるなど、並列計算を利用して計算性能を向上させることが必要となっている。ロボットが提供したいタスクは認識、検索、シミュレーション、推定などの高度な処理からなることが多く、実環境における大規模な問題を高速に解くために、強力な並列計算環境を利用したいという要求は強い。並列計算のためのプログラミング環境として、主に科学技術計算に用いられている MPI や OpenMP、あるいは分散オブジェクト環境などの並列言語処理系が提案されているが、これらは必ずしもロボットプログラミングにおける要件を満たしてはいない。

我々は、タプルスペースによる通信同期機構を用い、ロボットプログラムに頻出するイベントドリブンなタスクを簡単に記述できるような分散計算環境 “coalsack” を提案した。本稿では、coalsack の開発経験を紹介するとともに、近年提案され広がりつつあるロボットプログラミング用の言語処理系をいくつか紹介し、それらの設計を俯瞰することでロボットプログラミング用言語に求められている機能要件を明らかにする。

2. coalsack

2.1 ロボットプログラミングでの頻出構造

ロボットに求められる仕事、特に人間を対象としたタスクにおいて、その特徴は高いインタラクティビティにある。日常生活の中で求められるのは、数日間にわたってひたすら同じ動作を繰り返すようなタスクだけ

†1 東京大学
The University of Tokyo

```
A
take_picture(picture) {
  feature = sever1.sift(picture);
  server2.evaluate(feature);
}
-----
B
take_picture(picture) {
  feature = sever1.sift(picture);
  server3.logging(picture);
  server2.evaluate(feature);
}
```

図 1　分散オブジェクトモデルによるタスク記述

ではなく、人間の要求に応じて、即座に多様な動作を実行するタスクも多い。ロボットプログラミングにおいては、このようなタスクが記述しやすい言語が要求される。

インタラクティブな処理を実現する際には、イベントドリブンなタスク記述を用いることが多い。ユーザからの要求、センサ入力の変化などをきっかけにイベントが起き、必要なサブタスクが次々と実行されて、全体としてロボットアプリケーションが動作する、という構造である。coalsack はこのような記述を容易に実現できることを目指して設計を行った。

2.2 対象問題

coalsack の扱うべき対象問題には以下の特徴がある。

依存関係グラフ

対象問題全体は、依存関係を持つ多数の小問題から構成される。小問題それぞれは、必要なデータやイベントを待ち、必要な処理を行い、新たなデータを作成して別のタスクに受け渡していく。センサデータを収集して動作を行う場合など、ロボットプログラミングには典型的に現れる構造である。

タスク構造の変更

新たなセンサを付け加えてロボットの行動範囲を広げる、既存のセンサデータを入力とする新たなアプリケーションを導入する、など、タスクの依存関係グラフの変更を柔軟にかつ簡単に扱えることが望ましい。

例えば、分散システムのプログラミングで広く使われている分散オブジェクトモデルによる記述を考えてみる。**図 1** のコード A は、`picture` というデータ入力に対し、`sift()` という処理を加え、その結果に `evaluate()` という処理を加える、というタスク依存関係を分散オブジェクトモデルで記述したものである。

```
("raw_data", ?picture) ->
    feature = sift(picture)
    coalsack.out("feature", feature)
```

図 2　ルールによるタスク記述

ここで、新たなタスク `logging()` が依存関係に加わると (コード B)、タスクを呼び出す側のコードを変更し、`logging()` を付加しなければならない。つまり、先行タスクは次に呼ばれるべき全てのタスクを知っている必要があり、ダイナミックにタスクグラフが変更される時にはうまくモジュール化できないことになる。

並列処理の適用

PC クラスタなどを用いた並列・分散処理が利用できる環境は多い。これらの環境での分散プログラミングでは、通信、同期、負荷分散など多くの特有の処理を扱わねばならず、プログラミングを難しくしている。これらの処理を隠蔽し、容易にかつ効率よく並列処理を適用できることが求められる。

タスク実効場所の容易なコントロール

大量のセンサデータを前処理する部分はセンサに近い計算機で行いたい、ロボットの制御に関する処理は機構からのレイテンシが短いロボット内部の計算機で行いたい、など、タスクを処理する計算機を制御したい局面は多い。タスクと計算機のマッピングを柔軟に制御できることが求められる。

2.3 設　　計

coalsack では、イベントドリブンな処理を、タスクが実行される条件を記述する「ガード部」と、その条件が満たされたときに実行される処理「アクション部」からなる、「ルール」として記述することができる。**図 2** にルールによるタスク記述の例を示す。`->` の前の部分がガード部、後ろがアクション部であり、ガード部は発火の条件となるタプルを記述している。

coalsack はタプルスペース[1] に似た通信同期機構を持つ。図 2 のガード部は、2 要素のタプルで、1 要素目が`"comp_sift"`という文字列であるもの、にマッチすることを示している。この条件を満たすタプルが coalsack のタプル空間に存在するとき、ルールに示したアクション部が実行されることになる。アクション部には逐次プログラムを記述する。ここでは、現在の coalsack の実装言語である Python によるコードが記述可能である。アクション部の 2 文目、`coalsack.out()` は、2 要素のタプル `("feature", feature)` をタプル空間に書き込むことを意味し、このタプルによって対応する新たなルールが発火する、という形式でイベ

```
A
tuplespace.in("raw_data", ?picture)
feature = sift(picture)
tuplespace.out("feature", feature)
----
B
tuplespace.in("raw_data", ?picture)
feature = sift(picture)
tuplespace.out("raw_data2", picture)
tuplespace.out("feature", feature)

tuplespace.in("raw_data2", ?picture)
logging(picture)
----
C
("raw_data", ?picture) ->
    logging(picture)
```

図 3 タプルスペースの機能だけを用いたとき

ントドリブンなプログラムが記述できる。

ガード部のタプルの 2 要素目、`?picture` はマッチしたタプルの 2 要素目を変数 `picture` に束縛し、アクション部でその変数が参照できる。アクション部の 1 文目でこの `picture` を引数に通常の関数 `sift()` を呼び、結果の `feature` を得ている。

また、coalsack は分散オブジェクト空間を提供する。タプルの要素に Python の任意のオブジェクトを入れることが可能であり、アクション部 `coalsack.out()` で書き込まれたタプルの `feature` は、後続のルールが処理される計算機が別のものであってもその違いを意識することなく、計算に利用することができる。

タプル空間にタプルが書き込まれると、すべてのルールのガード部が検査され、条件にマッチするすべてのルールが発火し、対応するアクション部が実行される。またこの時、元のタプルは消える。

タプルスペースの機能だけで図 2 を記述すると**図 3** のコード A のようになる。ここで、`picture` に対して `logging()` 処理を追加すると、コード B のように既存コードを修正しなければならず、モジュール性が保たれないことになる。一方、coalsack のルール記述を用いると、コード C のルールを追加するだけで既存コードの修正は必要がない。

ルールにはそれぞれ、どこで action を実行するかという「場所」を指定するキーワードを付加することができる。計算機は、coalsack が提供する共有タプル空間に参加するとき、その計算機の提供する「場所」を示すキーワードを指定することができる。**図 4** の例は、データ `("raw_data", picture)` が入力されたとき、`pre_comp` というキーワードで指定される計算機で `comp_feature(data)` が処理され、その結果を `finish` という計算機で利用して `evaluate(feature)` する、という処理を表している。

```
@pre_comp
("raw_data", ?picture) ->
    feature = comp_feature(picture)
    coalsack.out("feature", feature)

@finish
("feature", ?feature) ->
    evaluate(feature)
```

図 4 ルールと実行場所

複数台の計算機に対し同じ場所キーワードを持たせることができる。この場合、発火したルールはそれらの計算機のどれか 1 台で実行される。それぞれの場所において、タプルはキューで管理されており、その場所に属する計算機は自身がアイドル状態になったらキューから 1 つタプルを取得し、対応するアクションを実行する、という動作が行われる。図 4 において複数の `("feature", x)` というタプルが出現した場合、`evaluate(feature)` は `feature` というキーワードを持つ計算機群のうち、実行が可能になったものに次々と割り当てられるという、動的負荷分散が行われる。

2.4 実　　装

実装に際して、Python を用いた分散オブジェクト環境処理系 gluepy[2](の開発版) を用いた。タプルにはこの処理系で利用可能な分散オブジェクトを入れることが可能である。

実装は単純なマスタワーカ方式をとっている。1 つのマスタが全てのタプルを管理し、対応するルールを選択し、行うべきタスクをキューで管理する。その他の計算機群はワーカとなり、マスタの指示にしたがってタスクを処理し、タスク実行が終わったらマスタにその旨を伝え、タスクを待つ。

2.5 デモシステムによる評価

coalsack 上に思いだし支援のための日用品データベースシステム[3] を作成し、小規模な実験環境での動作確認を行った。

このシステムは、部屋の数ヶ所にある引き出しの上の天井にカメラを設置し、引き出しに物をしまう時の

画像を取得する。画面に変化があったときのみ画像を取得することで保存データの量を抑制し、引き出しが開いている期間を秒間 3 フレーム程度で撮影する。この画像から、システムに登録された日用品を検出し、日用品発見の場所と時刻を DB に登録する。この DB を用いることで、「あの電卓をしまったのはどこだったか?」という問いに対し、「最後にこの引き出しで見ました」と答えることができる。

物品検出では SIFT 特徴量[4] による類似点検出を用いた。1 個の類似点が発見される確率と画像中の類似点分布から、簡単なモデルに基づく物品の存在確率を求めることにした。

タスク全体を、画像取得、ストレージへの移動と DB への登録、SIFT 特徴量抽出、テンプレート画像との特徴量マッチング、物品存在確率への変換と結果の DB 登録、という小タスクに分解し、画像が入力されるたびにこれらが順に実行されるようなタスクグラフを coalsack によって構築した。SIFT 計算や類似度計算の部分は C++で逐次プログラムにより記述したモジュールを呼び出し、coalsack はタスクのコントロールや通信部分を担当する、タスク並列を実現した。

既知物品 20 個程度、引き出し 20 個、カメラ 5 台からなる実験環境を作成し、デモシステムを構築した。既知物品のテンプレート画像は、物品の向き、距離などの条件を変えた様々な画像を用意するため、50 枚程度となった。計算環境には dual CPU, quad core Xeon 2.33GHz, 16GB memory のサーバ 5 台から構成される小規模 PC クラスタを用いた。

環境カメラから取得される画像は 1280x1024 ピクセルの高解像度であり、この計算機で SIFT 計算を行うと 1 画像あたり 5 秒程度の計算時間が必要となる。また、テンプレート画像との対応点探索には、1 ペアあたり 0.5 秒程度が必要であった。入力画像ごとに独立して計算することでこれらの処理を並列化し、実世界の変化が DB に登録されるまで 15〜30 秒程度の遅れで追従できるようなシステムを構築できた。

3. 関 連 研 究

3.1 Robotics Studio

Microsoft 社が提供するロボット向け開発環境に Robotics Studio[5] がある。Robotics Studio は分散オブジェクトモデルによるプログラミング環境を提供する。オブジェクト間通信には SOAP を用いており、実装は.net による。C#、Python、C++等の言語バインドが利用できる。

オブジェクト間をリンクで接続してビジュアルプログラミングするツールや、動作を物理シミュレーションすることが可能なシミュレータも備えた、統合開発環境を提供している。

3.2 RT middleware

RT middleware[6] は、産総研が中心となって提案しているロボットのためのプログラミング環境規格である。腕、台車などのロボットの部品をコンポーネント化し、自由に組み合わせて使えるようにすることで、ロボット開発の効率化を目指したものである。実装の一つに、産総研による OpenRTM がある。

分散オブジェクトモデルを採用しており、CORBA を使用している。内部に状態遷移マシンを持つ「コンポーネント」を組み合わせる、というプログラミングスタイルを提供する。

3.3 ROS

Willow garage 社が提供するロボット用プログラミング環境が ROS[7] である。分散した “node” と呼ばれるプロセスが、お互いにメッセージを送りあってアプリケーションを実行するというモデルを採用している。

4. 考　　察

4.1 共通する機構

関連研究に挙げた 3 つのロボット用プログラミング環境は、次のような共通する構造を持つ。

- 分散環境
  ネットワークで接続された分散環境を利用可能である。
- プロセス群による処理
  内部にループを持つプロセス群を管理し、イベントドリブンなタスクを実行できる。
- 通信機構
  - イベント通知 (一方向性の通信)
  - Remote Method Invocation
  - データストリーム (イベント通知と同様だが、効率などが改善されたもの)

  の、目的の異なる 3 つの通信機構を提供している。使いやすい処理系のためには、このような通信機構が必要であると考えられる。

4.2 接続関係変更への対応

単純な RMI ではタスク間の依存関係、接続関係を柔軟に変更できないという問題に対し、以下の 2 つの方法で対応している。

通信路の両端を指定する

Robotics Studio、RTM では、通信主体であるプロセスに通信用の端点、“port” を作成し、port の組を指定することで通信路を形成する。これにより、新た

な接続を追加する場合にも、既存の通信路指定に影響は発生しない。

通信路に名前を付ける

ROS では、通信路を Topic と呼んでおり、それ自体に名前が付加されている。通信路にデータを publish すると、その Topic を subscribe しているプロセス全てに同じデータが届く。新たな接続が必要になったときは、新たなプロセスが subscribe するだけでよく、既存の通信路指定への影響は発生しない。

Robotics Studio にもデータを発生する port を subscribe する機構が備わっており、同様の仕組みを利用することができる。また、提案手法では、タプルの要素が通信路名に相当していると考えることができる。

4.3 提案手法との比較

関連研究では、通信手法に大きく 3 通りの機構を提供しており、目的に合わせて使い分けることができる。coalsack のタプルにもそれらの目的を踏まえた何らかの指定と最適化を入れる必要があると考えられる。

また、関連研究では通信主体となるのは内部状態を持ったオブジェクトであり、状態を持たないイベントハンドラだけが存在する coalsack とは異なっている。並列処理のために複数の計算機が同じ種類のタプルを処理するが、それら同じ「場所」にあるイベントハンドラ群で共有できる内部状態を提供するべきかもしれない。

一方、関連研究には分散計算を積極的にサポートする仕組みは存在しない。高度な認識計算など、分散計算によって高速に処理を行いたい場面は今後ますます増えると思われ、提案手法のような、通信・動機の隠蔽、動的負荷分散機構などを備えた処理系が必要になると考えられる。

4.4 ロボットプログラミング環境への要求

シミュレーションと実体

プログラムの開発時には、実際のロボットを用いるのではなく、物理シミュレータを利用してシミュレーション世界で動作を確認することが多い。センサ入力に関しても、デバッグ時点ではログデータを用いて再現性のある入力を与えたいことが多い。実体のロボットとシミュレータ、実体のセンサとログデータ、のようなマッピングを自由に切り替え、開発時に利用していたプログラムがそのまま実運用にも適用できるような構造になっていることが求められる。

Robotics Studio などの関連研究では、このような構造を目指している。

デ バッグ

デバッグにおいては、必要に応じて通信路に流れたデータを全てロギングし、アプリケーションの実行リプレイを行う、という手法をとることが多い。大規模分散システムのデバッグと同様の難しさがあると考えられ、ログの取り方や必要最小限の再実行の実現など、解決すべき課題は多い。

アプリケーション境界の管理

現在のところ、ロボットの開発環境は基本的に、世界に 1 つだけアプリケーションが動作していることを想定した開発スタイルをとっている。アプリケーションのためのプロセス群の開始、終了などのコントロールは一斉に行われることが想定されている。しかし、今後、既に分散環境上でロボットに関するアプリケーションが動作している状態で、新たなアプリケーションを追加したり、一部のアプリケーションを修正したりするなど、動的な環境を提供することが必要になってくると考えられる。

複数のアプリケーションを動作させるためには、アプリケーションごとに名前空間やコントロールの単位を分離する界面を導入し、界面ごとに管理できることが必要となる。プログラムの構成要素であるプロセス群と、ロボットやセンサなど実際のデバイスとのマッピングを、複数のアプリケーションでどのように破綻なく実現するか、など解決すべき問題が多数存在する。

ROS は名前空間に関して階層構造を許している。これを用いると、プロセスグループからなる複数のアプリケーションコンポーネントを名前衝突させることなく同時に導入することが可能になる。しかし、同一のプロセスを複数のアプリケーションから重複利用したい場合にはまだ問題が残る。プロセス群や通信路の名前管理、処理制御などを行う “master” ノードは ROS 空間に 1 つだけであり、この master をロボットごとに 1 つ置くのか、部屋ごとなのか、建物ごとなのかという問題、つまり master の管理領域をどの範囲にするのか、の決定に課題が残る。

5. おわりに

本稿では、ロボットアプリケーションに頻出するイベントドリブンなタスクを記述しやすいプログラミング環境 “coalsack” を提案し、その設計方針を示すとともに、思いだし支援用日用品データベースシステムに適用した事例について紹介した。

また、ロボットのためのプログラミング環境を紹介し、必要とされる通信機構、扱いやすいプログラミングモデルを概観するとともに、今後望まれる処理系の機能として、シミュレーション環境と実体との自由なマッピング切り替え、デバッグ手法の改善、アプリ

ケーション境界の導入の必要性、の 3 点にまとめた。

ロボットは今後身近なものになっていくと期待され、そのプログラミング環境は黎明期であるといえる。より多くの計算リソースをオンデマンドで利用したり、複数のアプリケーションを同時に動かすなど、より動的なプログラミング環境が要求されていくと想定され、管理単位の分離と実行効率の向上を狙うための新たな界面の導入が必要になっていくと考えられる。

謝　　辞

この研究は，文部科学省「先端融合領域イノベーション創出拠点の形成：少子高齢社会と人を支える IRT 基盤の創出」で行われたものである．

参 考 文 献

1) Gelernter, D.: Generative communication in Linda, *ACM Trans. Program. Lang. Syst.*, Vol.7, No.1, pp.80–112 (1985).
2) Hironaka, K., Saito, H., Takahashi, K. and Taura, K.: gluepy : A Simple Distributed Python Framework for Complex Grid Environments, *21st Annual International Workshop on Languages and Compilers for Parallel Computing (LCPC2008)*, LNCS, Vol. 5335, Springer-Verlag, pp.249–263 (2008).
3) 高橋桂太，横山大作，森　武俊，植田亮平，山崎公俊，岡田　慧，松本　潔：思い出し支援のための日用品データベースシステム，日本ロボット学会学術講演会，pp.2E2–04 (2009). 神奈川県横浜市.
4) Lowe, D.G.: Distinctive image features from scale-invariant keypoints, *International Journal of Computer Vision*, Vol. 60, No. 2, pp. 91 – 110 (2004).
5) Microsoft: Robotics Studio ホーム. http://www.microsoft.com/japan/robotics /default.mspx.
6) Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T. and Yoon, W.-K.: RT-middleware: distributed component middleware for RT (robot technology), *Proceedings of 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2005)*, pp. 3933–3938 (2005).
7) Willow Garage: ROS ホーム. http://www.ros.org/wiki/.

質 疑 応 答

Q (和田 英一) 発表の内容は、ロボットと人間のインターフェースとなっていないか? 人間を超えるような intelligence を前提とはできないか。

A 残念ながら今のところはまだそれほどの想定はできていない。あくまで人間のプログラミングのためのインタフェースを考察した。

Q (和田 英一) coalsack の名前の由来は。

A 暗黒星雲の「石炭袋」であり、中でイベントが多数発火して何かが生まれるイメージから名づけている。

Q (笹田 耕一) 性能について知りたい、何らかの制約は (特にリアルタイム性)

A 今のところリアルタイム性を必要とする部分に使えるほどのレイテンシではない。ロボット、センサ等、個々のデバイス間を簡単につないでプログラミングする、という用途を想定している。

Q (笹田 耕一) 「デバイス間」とは「距離がある」ということかと思うが、もっと近いものの間でのやり取りはどうか。

A 1 つのマスタが全ての処理をコントロールする、という方式だけではなく、「場所」ごとに制御単位を分けるなど実装を工夫することで対応できる可能性はある。

Q (竹内 郁雄) スライド『coalsack との比較 (2/3)』での「場所」は何を指すのか、「オブジェクト」のことではないか。

A 複数の計算機にまたがった実体であり、オブジェクトのように内部状態を持ったものではない。

Q (田中 哲) スライド『タプルスペースだけではうまくいかない』のはどううまくいかないのか。

A タプルスペースの機構だけだと、同じタプルを複数の in プリミティブの両方で待つことができないため、タスクの依存関係を変更する時に元のコードに修正を加える必要がある。

Q (田中 哲) イベントを待つものが複数いた場合には、「全員発火」してから消えることになるのか。その「全員」を管理する仕組みが必要であるということか。

A その通りで、1 つのマスターが全ての発火を管理している。

Q (寺田 実) 「あるイベントともうひとつのイベントを待つ」というようなものは書けるか? starvation を起こしそうなので、何らかのプリミティブが必要ではないか。

A 今は書けない。記述力を上げるために大切そうなので今後考えていきたい。