例外処理にともなうプログラムコード錯雑化問題を解決する構文を用いた並行例外処理の実現

尋木信一[†] 小田謙太郎[†] 吉田隆一[†]

プログラミング言語に例外処理機構を導入する際の問題の1つに，例外処理の記述にともなうプログラムの可読性の低下があげられる．さらに分散計算環境においては，より複雑な例外を取り扱う必要があるため，単一の計算環境における例外処理より高度な機構を提供することがプログラミング言語に要求される．本論文の目的は，分散計算環境に必要となる並行処理中の例外処理に適した並行例外処理モデルを提案することと，プログラムの可読性を損なわずに高い記述力を持つ例外処理構文を提案することである．分散計算環境上では複数のオブジェクトが互いに並行実行可能であるので，同時に複数発生する例外を取り扱う必要がある．そこで，我々の提案する並行例外処理モデルでは，そのような並行処理中に発生した複数の例外をそれぞれ個別に，あるいはいくつかまとめて1つの例外状態と見なして取り扱うという2つの方法をプログラマに提供する．さらに，我々の提案する例外処理構文を用いれば，プログラマは例外処理を通常のプログラムとは別ファイルに記述する**例外ハンドラクラス**として定義することが可能である．このようにして，例外処理の記述に起因するアプリケーションプログラムの可読性低下問題を解決するとともに，例外処理コードそのものの再利用が可能となる．本論文では，1つの適用例として我々が開発中の分散オブジェクト指向計算環境 Juice への導入について述べる．

Concurrent Exception Handling without Code Tangling Caused by Exceptional Behavior

Shin-ichi Tazuneki,[†] Kentaro Oda[†] and Takaichi Yoshida[†]

When introducing exception handling mechanism into a programming language, a question arises as to how to reduce the code tangling between codes for the application program domain and codes for detecting and handling exceptions. In a distributed computing environment, a programming language have to provide a more powerful exception handling facility in order to deal with complicated exception handlings. In this paper we propose a concurrent exception handling model for a distributed application and an exception handling facility that eliminates code tangling caused by exceptional behavior. It is necessary to deal with multiple exceptions in a distributed computing environment because exceptions in such environment may be raised simultaneously from concurrent active objects. In our model, programmers have the flexibility of either handling all exceptions individually or grouping all exceptions together and handling them as a single one. Furthermore, programmers can describe exception handlings by an *exception handler class* written in a file different than the application program file. Therefore, it gives advantages for reducing the code tangling caused by exceptional behavior and reusing codes related to exception handlings. Finally, a prototype that models our approach has been implemented into our distributed object-oriented computing environment called *Juice*.

1. はじめに

プログラムの実行中に発生した誤りや障害からの実行時における回復を行う目的で，例外処理機構が用いられる．実際に，多くのプログラミング言語（Lisp, C++, Java など）にも例外処理機構が導入されている．一方，分散オブジェクト指向計算環境では，複数のオブジェクトが並行に実行可能である．この場合，通常の逐次構文であれば例外を起こす可能性のある処理対象はつねに1つであるが，並行構文であればそれが複数存在することになる．そのため，逐次型の例外処理機構のモデルでは対応できず，より複雑な機構が必要となる．このように，分散計算環境における例外処理を実現するためには，並行処理中に発生する例外を取り扱う手段と，それをプログラミング言語から表

† 九州工業大学情報工学部
Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology

現する方法について考慮する必要がある．

また，プログラミング言語に例外処理機構を実現する際の問題の 1 つとして，プログラミングの複雑化があげられる．これは，プログラマがアプリケーションプログラムを記述する際に，本来のアルゴリズムとは無関係の例外処理コードを付加することが原因となる．さらに，これはプログラムの可読性の低下にもつながる問題である．

本論文の目的は次の 2 つである．第 1 に，分散計算環境に必要となる並行処理中の例外処理に適した並行例外処理モデルを提案することである．第 2 に，プログラムの可読性を損なわず，なおかつ提案する並行例外処理モデルを表現可能な例外処理記述モデルを提案することである．我々の提案する並行例外処理モデルは，Dijkstra の **cobegin-coend** 文[2]による構造的な並行記述モデルと同様のモデルであればそのまま適用できる．

我々の提案する並行例外処理モデルでは，並行処理中に複数発生した例外をそれぞれ個別に，あるいはまとめて 1 つの例外状態として取り扱うという 2 つの選択肢をプログラマに提供する．さらに，例外処理の記述にともなうプログラムの可読性の低下に関する問題を解決するために，例外処理コードを通常のアプリケーションプログラムから分離する．つまり，1 つのファイルには，純粋に本来のアプリケーションを実現するプログラムを記述し，別ファイルに例外処理コードを記述する．例外処理の内容は，アプリケーションに依存するだけでなくプログラマの方針にも依存する．このように別ファイルに記述することは，アプリケーションプログラムの本来の内容は変更せずに，異なる例外処理を適用した新たなアプリケーションを開発する場合に有効である．本論文では，1 つの適用例として我々が開発を進めている分散オブジェクト指向計算環境 Juice に並行例外処理機構を導入する．Juice は，我々が開発中の開放型分散オブジェクト指向計算環境を提供するシステムであり，動的な環境の変化に柔軟に対応することが可能である．

本論文の構成は，まず 2 章で我々が開発を進めている分散オブジェクト指向計算環境 Juice についての概要を説明する．次に，3 章で我々が提案する例外処理構文，および並行例外処理モデルについて述べる．4 章で，本論文で提案した並行例外処理モデルを Juice に導入するための実装方法について説明し，実装の評価を行う．5 章では，関連研究の紹介と本研究との比較について述べる．最後に，6 章で今後の課題について考察した後，7 章で本論文のまとめを行う．

2. 分散オブジェクト指向計算環境 Juice

2.1 Juice システム

分散オブジェクト指向計算環境 Juice は，オブジェクト指向プログラミング言語と分散計算環境を統合することを目指して我々が開発を行っているシステムである．その目的は，プログラマがプログラミング言語から直接的に分散アプリケーションを開発できるようになることである．また，Juice は動的な環境の変化に柔軟に対応するために *flying object* モデル[10]を採用している．Juice は，Juice 言語，トランスレータ，実行時環境から構成され，オペレーティングシステムによって提供されるネットワーク上に分散する複数のアドレス空間上に 1 つの仮想的なオブジェクト空間を提供する．

Juice 言語は，Java[11]の構文をベースとした分散アプリケーション開発向けのオブジェクト指向言語である．そこで，Juice 言語は Java の構文に加えて，分散計算に必要となる並行メッセージ送信構文を提供する．また，Juice システムを標準の Java VM 上で動作させるために，Juice 言語から Java へと変換するトランスレータによって実現する．

2.2 Flying Object モデル

分散計算環境を実現するためには，プログラマが考慮するアプリケーションレベルより低レベルで必要となる様々な機能を，システムがあらかじめ提供する必要がある．たとえばその 1 つの機能として，異なるアドレス空間に存在するオブジェクト（以降，リモートオブジェクトと呼ぶ）間のネットワークを介したメッセージ送受信の機構が必要である．その機構をプログラマから透過的に提供することで分散計算環境を実現している例として，Java RMI[12]や HORB[4]がある．Java RMI や HORB は，リモートオブジェクト間のメッセージ送受信を実現する機構を代理オブジェクトによって提供している．代理オブジェクトは，ユーザ定義クラスとは別クラスとして自動生成される．このように，アプリケーションレベルより低レベルの機構をプログラマに対して透過的に提供することで，分散計算環境を実現することが可能である．そこで我々は，我々の目指す分散オブジェクト指向計算環境に必要となる機能をモジュール化するために，flying object モデルを提案した．

Flying object モデルは，ネットワーク透過性を実現するとともに計算環境の変化に適応させるために，オブジェクトの振舞いを動的に変更する機構を提供する．Flying object は，次にあげる 4 つのオブジェク

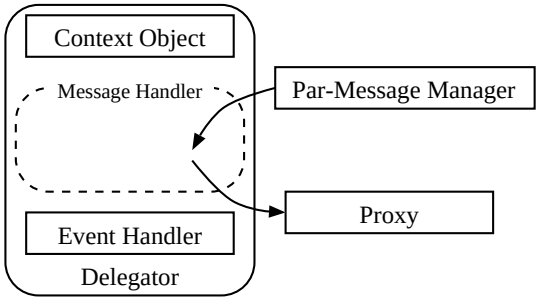


図 1 Flying object モデル
Fig. 1 The Flying object model.

トによって構成される（**図 1** を参照）.

- Context Object
- Message Handler
- Event Handler
- Delegator

*Context Object*は，ユーザが定義するオブジェクトそのものである．*Message Handler*は，システムの持つ様々な機能をモジュール化したオブジェクト群の総称である．たとえば，Java RMI や HORB の代理オブジェクトと同等の機能を実現する *Proxy*，並行メッセージ送信を実現する *Par-Message Manager*などがある．*Event Handler*は，環境の変化をとらえるモジュールである[8]．プログラマは，あらかじめ環境の変化に対する処理をクラスごとに記述することができる．Event Handler は，あらかじめ定義されているイベントが発生した場合，それに対して定義されている処理を行う．Juice 上のユーザ定義オブジェクトは，この Message Handler や Event Handler を利用することで様々な環境に適応することができる．

前述した Java RMI や HORB と同様に，Juice システムが提供する Message Handler や Event Handler のような機構をプログラマに対して直接提供することで，分散計算環境を実現することは可能である．しかし，この方法ではプログラマがアプリケーションとは直接関係のない低レベルなモジュールを明示的に利用する必要がある．たとえば，Java RMI や HORB では，リモートオブジェクトへのアクセスを行う場合，自動生成された特殊なクラスから代理オブジェクトを明示的に生成する必要がある．このように，プログラマにとってはプログラマ自身が定義していない特殊なクラスを意識して利用する必要があるため，この方法ではオブジェクトへのアクセスに対する透過性は実現されていないといえる．そこで，Juice ではその問題を解決するために *Delegator*を提案した．Delegator の導入は，前述した Message Handler や Event Handler のようなシステムが提供する機能をユーザ定義オブジェクトに導入しても，見かけ上はユーザ定義オブジェクトと同様に見えるようにすることが目的である．Delegator は，Context Object，Message Handler，Event Handler を内包するためのオブジェクトである．Delegator は，ユーザ定義オブジェクトである Context Object と同一クラスとなるようにトランスレータによって自動生成される．Delegator に内包される Message Handler の種類によって，ユーザ定義オブジェクトと同一クラスでありながら異なる振舞いをするオブジェクトを生成することが可能である．このようにして，プログラマはリモートオブジェクトへのアクセスも，同一アドレス空間に存在するオブジェクトへのアクセスと同じ方法で行うことができる．

```
Par-block-statement :
    par {
        < message-sending-statement-list >
    }
Async-qualifier :
    async < message-sending-statement >;
```

図 2 par ブロック文と async 修飾子
Fig. 2 The par block statement and the async qualifier.

また，flying object モデルでは，Message Handler を様々な機能を持つモジュールに変更することで動的な環境の変化に対応することができる．例外処理を実現する機構はプログラムの実行が例外コンテキスト中にあるときのみ必要となるので，あらかじめ静的にシステムに組み込むと計算資源の浪費につながる．そこで，flying object モデルのようにオブジェクトの持つ機能を動的に変更できる特性は，例外処理機構のようにプログラムの実行が特定のコンテキスト中にあるときのみ必要となる機構を実装するのに有効である．

2.3 Juice 言語における並行記述

Juice 言語は Java をベースとしているため，オブジェクトの振舞いは Java と同じ方法で記述できる．さらに，Juice が提供するオブジェクト空間上での分散計算をサポートするために，Java の構文を拡張して 2 種類のメッセージ送信構文（**par** ブロック文と **async** 修飾子（**図 2** を参照））を追加している．プログラマは，これらの構文を利用してプログラミング言語から並行計算を直接表現することができる．**par** ブロック文は，ブロック内に複数のメッセージ送信構文を含んだ構造的な構文である．**par** ブロック内のメッセージは並行に送信され，それらの返答がすべて受け取られた後，ブロック文の次の文が実行される．また，

async 修飾子を用いたメッセージ送信が実行されると，その返答を待たずに次の文が実行される．返り値のあるメッセージ送信が **async** によって修飾されると，その返り値が格納される変数が参照されるときに同期がとられる．

本論文では，特に **par** ブロック文による並行メッセージ送信時の例外処理に注目して並行例外処理モデルを提案し，Juice 言語でそれを表現する方法について述べる．

3. 並行例外処理モデル

3.1 例外と例外コンテキスト

例外処理機構を実現するためには，まずプログラミング言語で例外を表現する必要がある．一般的に，オブジェクト指向言語では例外をクラスとして表現する．クラスは，クラス間の関係を表現する階層構造を形成することによって複数の他のクラスを抽象化できる．この性質を利用することによって，多数存在する例外を抽象的に取り扱うことが可能である[3]．そこで，Juice 言語でもその利点を活かすために例外をクラスで表現する．Juice は Java によって実装されているので，Java が提供する例外クラス階層をそのまま利用可能である．

また，プログラマがプログラミング言語上で例外を取り扱うためには，例外コンテキストを表現するための手段が必要である．例外コンテキストとは，同じコンテキスト中で発生した同じ例外に対しては同一の処理を行うということを表す領域である．既存のプログラミング言語では，例外コンテキストとしてオブジェクト単位，メソッド単位，あるいはさらに小さい領域を表現するためにブロック文のような構造的な構文を提供している．さらに，例外コンテキストごとに，そのコンテキストに適用する例外処理の戦略を表現する手段が必要である．ここでいう例外処理戦略とは，ある例外コンテキストで捕捉すべき例外とその例外の発生時に行う回復処理の組合せのことである．たとえば，Java では例外コンテキストを **try** ブロック文，捕捉する例外とそれに対する回復処理を **try** ブロック文に続く **catch** 節で表現する．このように例外処理コードの部分を **catch** 節にまとめて記述可能であるので，例外処理コードの部分をプログラムの通常のアルゴリズムの部分からある程度分離することができる．しかし，この方法では **try** ブロック文ごとにそれぞれの例外処理戦略を定義するため，例外処理コードがアプリケーションプログラム中に点在することになる．したがって，この方法では，まだ例外処理の記述に起因するプログラム可読性低下の問題を残している．そこで，Juice 言語では通常のプログラムコードと例外処理コードをそれぞれ別ファイルに記述することでこの問題を解決する．

Complain-block-statement **：**
complain(< *expression* >) {
< *statement-list* >
}

図 3 complain ブロック文
Fig. 3 The complain block statement.

3.2 例外処理に関する Juice 言語仕様

例外処理をプログラミング言語から表現するために，Juice 言語に **complain** ブロック文と例外ハンドラクラスを導入する．

complain ブロック文は，例外コンテキストを定義する構文である．**complain** ブロック文の構文を**図 3** に示す．**complain** ブロック文は，キーワード **complain** に続く小括弧内の `<expression>` に例外ハンドラオブジェクトが入り，中括弧内の複数の文を包含するブロックが例外コンテキストとなる．このようにして，その例外ハンドラオブジェクトを例外処理戦略として現行の例外コンテキストに対応付ける．図 3 中の `<statement-list>` には，通常処理のためのコード群が入る．このコード群を実行中に例外が発生した場合，対応付けられた例外ハンドラオブジェクトが示す例外処理戦略に従って例外処理が行われ，その後ブロックを途中終了する．

例外ハンドラクラスは，例外処理戦略を記述するためのクラスである．例外処理戦略に関する記述を例外ハンドラクラスという形にモジュール化することで，アプリケーションプログラムから分離する．例外ハンドラクラスは，例外メッセージ受信時の振舞いが定義された receive メソッドを持つ．receive メソッドは，発生した例外オブジェクトとそれを捕捉したオブジェクトを引数とする．つまり，Juice では，例外ハンドラオブジェクトへの例外メッセージ送信（receive メソッド呼び出し）の形で例外発生の機構を実現する．

また，Juice はあらかじめ例外ハンドラクラスとして **ExceptionHandler** クラス（**図 4** を参照）を提供する．プログラマが独自の例外処理戦略を定義するためには，**ExceptionHandler** クラスのサブクラスとして新たな例外ハンドラクラスを定義し，receive メソッドをオーバライドする必要がある．プログラマは，特別な例外処理戦略を必要としない場合，**ExceptionHandler** クラスを利用することができるの

```
public class ExceptionHandler {
    public void  receive(Object handlingObj, Exception excp) {
        System.out.println( excp + " has been raised." );
    }
}
```

図 4 例外ハンドラクラス ExceptionHandler
Fig. 4 The ExceptionHandler class.

で，新たに例外ハンドラクラスを定義する必要がない．**ExceptionHandler** クラスは，例外が発生する可能性のある処理対象が 1 つである場合に対応する例外処理戦略を表す．その例外処理戦略は，あらゆる例外を捕捉して，その例外名を含むエラーメッセージを出力するという内容である．

ここで例として，**図 5** (a) で示すような自動窓口機（ATM）のアプリケーションプログラムを考える．**ATMClient** クラスは，自動窓口機の機能である引き出し，預け入れ，残高照会，振り込みなどを実装するメソッド withdraw，deposit，balance，transfer などを持つ．このアプリケーションはクライアントサーバモデルを採用しており，**ATMClient** オブジェクトは，銀行側のシステム内に存在する **ATMServer** オブジェクトとネットワークを介した通信を行う．**ATMClient** クラスの複数のメソッドが，リモートオブジェクトである **ATMServer** オブジェクトと通信する必要があるため，それらすべてのメソッドでネットワークに関する例外が発生する可能性がある．**try-catch** 構文による例外処理の記述を行う場合，たとえその例外処理の内容が同じであったとしてもそれらすべてのメソッドに例外処理コードをそれぞれ記述する必要がある．しかし，Juice 言語では，それぞれのメソッドには **complain** ブロック文のみを記述し，例外処理は例外ハンドラクラスの receive メソッドにまとめて記述できる．

図 5 (a) の例外ハンドラクラスである **NetworkExceptionHandler** クラス，および **ATMExceptionHandler** クラスを図 5 (b) に示す．**NetworkExceptionHandler** クラスは，ネットワークを利用した通信に関する例外のための例外処理戦略が定義された例外ハンドラクラスである．Juice 言語では，**complain** ブロック文を用いてこの **NetworkExceptionHandler** クラスを例外処理戦略として採用すれば，ネットワークに関する一般的な例外に対処することができる．

また，この例の場合，ネットワークに関する例外に加えてこのアプリケーション独自の例外が考えられる．たとえば，残高額より多い金額を引き出そうとした場合（図 5 (b) 中の **InsufficientFoundsException** クラス）や，誤った口座を指定した場合（図 5 (b) 中の **WrongAccountException** クラス）などがある．その場合，図 5 (b) に示すように **NetworkExceptionHandler** クラスを継承して，**ATMExceptionHandler** クラスとして新たに考えられる例外に対する処理のみを定義するだけで済む．このように，Juice 言語では例外処理戦略を例外ハンドラクラスとしてモジュール化することによって，例外処理コードそのものの再利用が可能となる．

したがって，Juice 言語の例外処理記述は，Java の **try-catch** 構文と比べて次の 3 つの点で優れていると考える．

1 つ目は，記述量が少なくて済む点である．これは，

```
1 : public class ATMClient {
2 :     .....
3 :     public int withdraw( int money ) {
4 :         ....
5 :         complain( new ATMExceptionHandler() ) {
6 :             return atmServer.withdraw( money );
7 :         }
8 :     }
9 :     public void deposit( int money ) {
10:         complain( new ATMExceptionHandler() ) {
11:             ....
12:         }
13:     }
14:     public void balance(...) { .... }
15:     public void transfer(...) { .... }
16: }
```

(a) A code fragment of an application program

```
1 : public class NetworkExceptionHandler
2 :                         extends ExceptionHandler {
3 :     public void receive(Object handlingObj,
4 :                             ServerException e) { .... }
5 :     public void receive(Object handlingObj,
6 :                         NoSuchObjectException e) { .... }
7 :     ....
8 : }
9 : public class ATMExceptionHandler
10:                 extends NetworkExceptionHandler {
11:     public void receive(Object handlingObj,
12:                         InsufficientFoundsException e) {
13:         System.out.println( e.getMessage() );
14:     }
15:     public void receive(Object handlingObj,
16:                         WrongAccountException e) { .... }
17:     ....
18: }
```

(b) Codes for exception handlings

図 5 自動窓口機に関するアプリケーションプログラムの例
Fig. 5 Example of an application program for an automated teller machine.

前述したように，**try-catch** 構文ではそれぞれのメソッドに例外処理を記述しなければならないのに対し，Juice 言語では例外ハンドラクラスに例外処理をまとめて記述できるからである．

2つ目は，例外処理内容の変更や修正を容易に行うことができる点である．これは，同じ処理内容の例外処理コードもそれぞれのメソッドに記述する **try-catch** 構文とは異なり，Juice 言語では例外ハンドラクラス1カ所に記述するためである．Juice 言語では，例外ハンドラクラス1カ所の変更が，その例外ハンドラクラスを例外処理戦略として用いるすべての **complain** ブロック文に反映される．

3つ目は，例外処理コードそのものが再利用可能な点である．Juice 言語では，例外処理を例外ハンドラクラスに記述するので，通常のクラスと同様に再利用可能である．前述したように，**complain** ブロック文に既存の例外ハンドラクラスを用いるだけで，例外処理コードそのものを再利用できる．また，例外処理戦略は，同じアプリケーションに対しても各々のプログラマの方針によってその戦略の内容が異なる場合がある．このような場合，**try-catch** 構文のようにアプリケーションプログラムに直接例外処理コードを記述する言語では，アプリケーションのアルゴリズムは同じであるにもかかわらず，その一部の例外処理コードの違いからアプリケーションプログラムを再利用できない．しかし，Juice 言語では通常処理のプログラムと例外処理のプログラムを別ファイルに分離して記述するため，それぞれのプログラマが例外ハンドラクラスのみを各自の要求に適したように再定義することができる．このようにして，アプリケーションプログラム自体には手を加えず，例外処理戦略のみが異なるアプリケーションを容易に開発することが可能である．

3.3 Juice における並行例外処理

並行処理中の例外処理としては，例外を起こす可能性のある処理対象が1つ以上あることを考慮する必要がある．そこで，そのような場合に発生する複数例外に対する例外処理戦略を表現可能な手段が必要である．Juice 言語では，1つの receive メソッドに複数の例外クラスを引数として定義することで複数例外に対する例外処理戦略を表現する．引数として複数の例外クラスを定義された receive メソッドは，それらすべての例外が発生したときのみ実行される．このような方法で，Juice 言語では複数例外に対しても，それらをまとめて取り扱うことが可能である．

また，並行メッセージ送信構文 **par** ブロック文によって表現される並行計算モデルでの例外処理においては，プログラマからの要求として次の2つが考えられる[13]．

(1) 並行処理中のすべてのオブジェクトの実行終了後，例外処理へ移行する．
(2) 並行処理中の1つのオブジェクトから例外が発生した時点で，ただちに例外処理へ移行する．

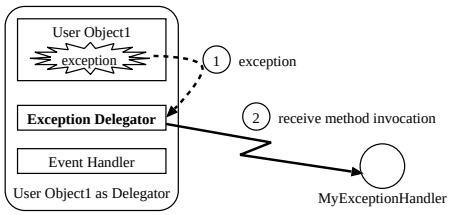
これら2つの要求はアプリケーションに依存するので，プログラマがアプリケーションに応じて選択的に利用可能であるのが望ましいといえる．また，これら2つの要求は，プログラマが例外処理戦略を例外ハンドラクラスとして定義する際に生じると考える．そこで，Juice ではプログラマが独自の例外ハンドラクラスを定義する際に親クラスとして使用するクラスの型によって，前述の2つの並行例外処理モデルの選択を行う．Juice では，前述した例外ハンドラクラスである **ExceptionHandler** クラスのほかに，そのサブクラスとして **MultipleExceptionHandler** クラスと **PreemptiveExceptionHandler** クラスを提供する．プログラマは，前述の要求 (1) で示す並行例外処理モデルを利用した独自の例外処理戦略を定義する場合，**MultipleExceptionHandler** クラスを親クラスとした例外ハンドラクラスを定義する必要がある．また，前述の要求 (2) で示す並行例外処理モデルを利用する場合は，**PreemptiveExceptionHandler** クラスを親クラスとした例外ハンドラクラスを定義する必要がある．このようにして，Juice 言語では2つの並行例外処理モデルの選択肢をプログラマに提供する．

4. Flying Object モデルによる並行例外処理機構の実装

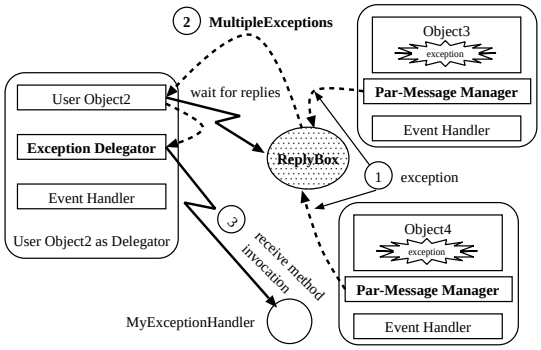
この章では，前章で述べた例外処理モデルを Juice 上に実現するために実装した *Exception Delegator* と Par-Message Manager について説明する．また，実装したシステムの有用性を示すために，実装の評価を行う．

4.1 Exception Delegator

例外ハンドラオブジェクトによる例外処理を実現するために，Exception Delegator を実装する．Exception Delegator は，Context Object 内で発生した例外を捕捉し，その例外を適切な例外ハンドラオブジェクトへ例外メッセージとして転送するオブジェクトである．Exception Delegator には，一度 Java の例外処理機構で発生した例外を捕捉し，それを例外メッセージとして適切な例外ハンドラオブジェクトへ転送する機能が必要である．Exception Delegator の振舞いの具体例を**図 6** (a) に示す．この例には，flying object と



(a) An example of single exception handling



(b) An example of concurrent exception handling

図 6 Exception Delegator と Par-Message Manager
Fig. 6 The Exception Delegator and the Par-Message Manager.

して生成されたユーザ定義オブジェクトとそれに対する例外処理戦略を定義した **MyExceptionHandler** オブジェクトの 2 つがある．ユーザ定義オブジェクト内で発生した例外は，それに組み込まれている Exception Delegator へ渡された後，その例外コンテキストに対応する **MyExceptionHandler** オブジェクトへ Exception Delegator による例外メッセージ送信が行われる．

前章で述べたように，Juice システムは独自の例外処理機構を備える Java による実装を行っているので，例外はすべて Java の例外処理機構で発生する．したがって，例外ハンドラオブジェクトによる例外処理を行うためには，その例外を一度 Juice システムで捕捉しなければならない．このような機能は，プログラムの実行が例外コンテキスト中にあるときのみ必要となるので，あらかじめ静的に flying object に組み込むと計算資源の浪費につながる．そこで，このような機能を動的に flying object に組み込むために，Exception Delegator は Message Handler として実装する．具体的には，トランスレータによるソースコード変換によって実現する．その例を**図 7** に示す．図 7 中の変数 `_mh` は，トランスレータによって付加された変数で，ユーザ定義オブジェクトに組み込まれている Message Handler オブジェクトへの参照を表す．図 7 の 2

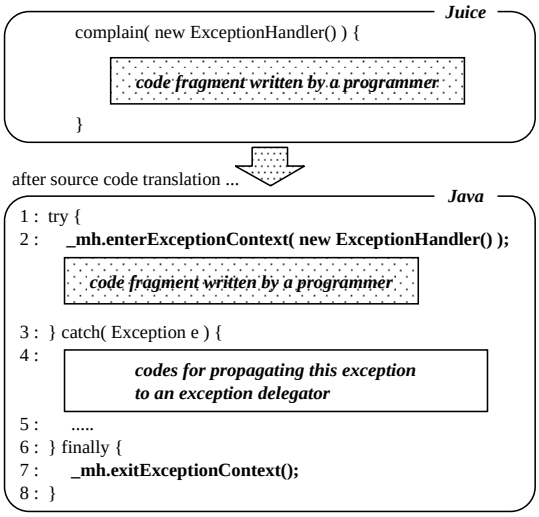


図 7 例外処理に関するトランスレータの生成コードの例
Fig. 7 An example of source code translation.

行目は，Exception Delegator をユーザオブジェクトの Message Handler に組み込むためのコードである．図 7 中の **try-catch** 文は，Java の例外処理機構で発生するすべての例外をとらえて，実際に発生した例外を Exception Delegator へ伝播させるためのコードである．さらに，**try** ブロックを終了する際に必ず実行される **finally** 節には，Exception Delegator を取り除くコードをトランスレータによって生成する．このように，Exception Delegator はシステムによって動的に flying object へ組み込まれるので，プログラマが Exception Delegator 自身を意識する必要はない．

4.2 Par-Message Manager

我々の提案する並行例外処理モデルを実現するために，Par-Message Manager を実装する．Par-Message Manager は，次にあげる 2 つの機構を備える．

第 1 に，**par** ブロック文のセマンティクスを実現するための機構である．Par-Message Manager は，ブロック内に記述されたすべてのオブジェクトへ並行メッセージ送信を行う．このとき，それぞれのオブジェクトからの返答を随時格納するための **ReplyBox** オブジェクトを生成する（図 6 (b) を参照）．並行メッセージ送信側のオブジェクトは，**ReplyBox** オブジェクトを介して返答を受け取る．このとき，並行メッセージ送信側のオブジェクトはすべての返答が到着するまで待つ．この **ReplyBox** オブジェクトを介したやりとりはプログラマから透過的に行われるので，プログラマが **ReplyBox** オブジェクトを意識する必要はない．

第 2 に，並行例外処理を実現するための機構である．返答待ちの間に例外が伝播されてきた場合，あらかじ

めプログラマによって設定されている並行例外処理モデルに従った振舞いをする．実際に発生した例外は，**ReplyBox** オブジェクトを介して送信側のオブジェクトへ伝播し，最終的に Exception Delegator へと渡される．このとき，前章で述べた要求 (1) であるすべての返答を待った後例外処理を行うモデルの場合，複数の例外が発生している可能性がある．Java の例外処理機構は同時に複数の例外を取り扱えないため，Exception Delegator へは例外 **MultipleExceptions** として渡される．**MultipleExceptions** クラスは，実際に発生したすべての例外を自身のインスタンス変数として包含する．Exception Delegator は，例外 **MultipleExceptions** から実際に発生した例外を取り出すことで，適切な例外ハンドラオブジェクトを呼び出すことができる．また，例外が発生した時点でただちに例外処理を行うモデルの場合，発生した例外がただちに **ReplyBox** オブジェクトを介して送信側オブジェクトへと伝播する．したがって，その時点でまだ実行中のオブジェクトからの返答はすべて無視される．

Par-Message Manager も Exception Delegator と同様にトランスレータによって動的に flying object に組み込まれる．具体的には，トランスレータは **par** ブロック内のすべての flying object に対して，Par-Message Manager を組み込むためのコードを生成する．Par-Message Manager の例外発生時の振舞いは，プログラマが例外コンテキストごとに採用する並行例外処理モデルに依存して異なるので，Par-Message Manager には現行の並行例外処理モデルの情報が必要である．これは，例外ハンドラクラスの型によって判断することができる．具体的には，現行の例外コンテキストに設定されている例外ハンドラクラスが **MultipleExceptionHandler** クラスからの派生クラスであるか，**PreemptiveExceptionHandler** クラスからの派生クラスであるかによって並行例外処理モデルを決定する．このように，並行例外処理モデルは静的に決定可能であるので，この情報もトランスレータによるソースコード変換時に付加される．

4.3 実装の評価

今回実装した Juice システムと Java において，各々の例外処理に関する実行時間の比較を行う．実行時間の計測に使用したアプリケーションは，次の 2 つのクラスによって構成される．

- **ExceptionalCallee** クラス
- **MyException** クラス

ExceptionalCallee クラスは，calleeMethod メソッドを 1 つ持つクラスである．calleeMethod メソッドは，必ず例外 **MyException** を発生する．これらに加えて，Juice 言語で書かれたアプリケーションプログラムには，例外処理戦略として **MyExceptionHandler** クラスがある．**MyExceptionHandler** クラスは，例外 **MyException** の発生に対してエラーメッセージを出力するという例外処理戦略を表す．

表 1 Java と Juice の例外処理機構に関する実行時間比較
Table 1 Comparison of execution time between Java and Juice.

	(1) Java	(2) Juice+ try-catch 文	(3) Juice+ 提案モデル
例外処理 無（捕捉のみ）	約 24 μs	約 141 μs	約 270 μs
遅延		5.9 倍	11.3 倍
例外処理 有	約 89 μs	約 176 μs	約 318 μs
遅延		2.0 倍	3.6 倍

アプリケーションの内容は，**ExceptionalCallee** オブジェクトの calleeMethod メソッドを 10000 回繰り返し呼び出すのにかかる時間を測定するプログラムである．このアプリケーションを（1）Java,（2）例外処理を **try-catch** 構文で行った Juice,（3）提案した例外処理モデルを導入した Juice の 3 種類によるプログラミングを行い，それぞれの実行時間を測定して違いを考察する．これらのプログラムをそれぞれ 10 回実行した平均時間を測定し，1 回の calleeMethod メソッド呼び出しから例外処理を終了するまでにかかった平均時間を**表 1** に示す．さらに，純粋に通常処理と例外処理の間の切換えにかかる時間を測定した結果も示す．

表 1 からも分かるように，(1) の Java における例外処理切換えにかかる時間と比べて，(3) の提案モデルの Juice における例外処理切換え時間は約 10 倍遅くなった．今回の実装では，我々の提案する例外処理機構を実現するために低レベルな部分で Java の例外処理機構をそのまま利用する方法を採用した．したがって，単純に Java の **try-catch** 構文による例外処理を行った場合と比べて，以下の項目に関する実行時間分だけ遅延が生じる．

(1) 例外処理に入る前処理としての Exception Delegator，およびユーザ定義の例外ハンドラオブジェクトの生成
(2) Java の例外処理機構によって捕捉された例外を，Exception Delegator へ委譲するためのメソッド起動
(3) ユーザが定義した例外処理を実行するための適切な例外ハンドラオブジェクトの receive メソッド起動

ここで，表 1 下段のエラーメッセージを出力するという単純な例外処理を行った場合の実行時間について考える．この場合は，(1) の Java による例外処理に比べて (3) の提案モデルによる Juice の例外処理を行った場合，約 3 倍の遅延で収まった．これは，例外処理の切換えにかかる時間に比べて実際の例外処理内容の実行時間の方が非常に大きいからである．したがって，提案モデルによる Juice の例外処理において例外処理切換え時間に生じた約 10 倍の遅延は，実際の例外処理内容の実行時間に吸収され，全体の実行遅延としては問題ない程度まで小さくなると考える．

また実行時間の遅延が生じる別の理由として，表 1 の (1) と (2) の項目の比較からも分かるように，基盤となる flying object モデルの実装自体が影響している．Flying object モデルは，その実装に関してはまだまだ発展途上であるので，将来的に実行時間の遅延をある程度抑えることができると考える．ただし，今回の実装では，前述したように flying object モデルの特徴である動的な Message Handler の切換え機構を利用し，プログラムの実行が例外コンテキストに入ったときのみ例外処理実行機構を組み込む．したがって，例外が発生しない場合，今回実装した例外処理機構による実行速度への影響はほとんどない．

また，提案したモデルでは，オブジェクト指向の概念を導入することによりすべての例外状態をオブジェクトとして表現するので，例外が発生した分だけオブジェクトが生成されることになる．このことが原因で，メモリ利用効率などの実行時パフォーマンス性能が悪化する可能性がある．しかし，プログラマが複雑な例外処理を記述する場合，例外発生時の情報が不可欠である．そのような複雑な情報をうまく統合できるというオブジェクト指向の利点は，メモリ利用効率を犠牲にしてでも例外をオブジェクトとして取り扱うということへの十分な理由になると考える．

5. 関 連 研 究

この章では，本研究と関連研究の比較を行う．特に，それぞれの並行例外処理モデル，および例外処理の記述性に注目して比較する．

5.1 アスペクト指向プログラミング言語 AspectJ

AspectJ [1)]は，Java にアスペクト指向プログラミングの概念[9)]を導入したプログラミング言語である．アスペクト指向プログラミングとは，1 つのアプリケーションを複数の異なる視点からモデル化してプログラミングを行う手法であり，1 つのコンポーネントプログラミング言語と複数のアスペクトプログラミング言語によって構成される．コンポーネントプログラミングは，たとえばオブジェクト指向プログラミングのように対象となるアプリケーションをいくつかのオブジェクトとなるコンポーネントに分割して設計し，それらを組み合わせて 1 つのアプリケーションを構成する手法である．AspectJ においては，Java で書かれたプログラムがコンポーネントプログラムとなる．アスペクトプログラミングは，*crosscuttig concerns*と呼ばれる単純にモジュール形式にあてはめることのできない問題をアスペクトとして表現するための手法である．AspectJ によって開発されたアプリケーションプログラムは，1 つのコンポーネントプログラムと 1 つのアスペクトプログラムで構成される．これら 2 つのプログラムは *join point*に従って，*aspect weaver*によって結び付けられる．Join point とは，crosscutting concerns に影響されるプログラム中の場所であり，ステートメントレベルあるいはオペレーションレベルで表現される．つまり，プログラマが定義するアスペクトは各 join point に適応される．

例外処理も crosscutting concerns の 1 つと考えられるので，アスペクトとしてアプリケーションの本来のプログラムとは別に定義できる．したがって，AspectJ の例外コンテキストは join point に依存する．AspectJ では，例外コンテキストをオブジェクト全体，メソッド全体として取り扱うことができる．さらに，ある例外が捕捉されるという事象を join point として用いることもできる．

また，AspectJ は Java をベースとしたオブジェクト指向言語であるので，プログラマは並行プログラミングのために明示的にスレッドを利用する必要がある．スレッド内で発生した例外はスレッド内で閉じてしまうので，標準のままではスレッド間にまたがる例外を取り扱うことはできない．したがって，並行例外を取り扱うためには，プログラマが特別なコードを記述する必要がある．

5.2 オブジェクト指向並行言語 ABCL/1

オブジェクト指向並行言語 ABCL/1 [14)]は，メッセージ送信によって例外の発生，伝播を行う方法を採用する．ABCL/1 は，例外をメッセージとして取り扱うため，例外処理は通常のメッセージに対する処理の記述と同様に行う[5)]．つまり，並行処理中の複数の例外の発生は複数のメッセージが送信されることで表現される．したがって，複数発生した例外をそれぞれ個別に処理することが可能である．このように，例外処理も通常メッセージの受信に対する処理と同等に扱われるため，複数の例外をまとめて取り扱うという表現はで

きない．

また，ABCL/1 におけるメッセージには，その送信時に *reply destination*と *complaint destination*というパラメータを付加することが可能である．reply destination は，そのメッセージに対する返答先を表しており，プログラマが任意のオブジェクトを指定することができる．complaint destination は，そのメッセージの処理中に発生した例外を処理するオブジェクトを表す．したがって，メッセージ送信側が，例外が発生した場合の処理を依頼するオブジェクトをあらかじめ指定することが可能である．

5.3 並列オブジェクト指向言語 Arche

並列オブジェクト指向言語 Arche[7]は，並行処理中に発生した複数の例外を処理するために，*resolution function*[6]による方法を提案している．resolution function は，発生した複数の例外を引数として受け取り，返り値として 1 つの例外を返す関数である．プログラマは複数例外が発生する可能性がある場合，それら複数の例外を 1 つの例外に置き換える手続きを resolution function としてあらかじめ定義する必要がある．このように複数の例外を 1 つと見なした例外処理の記述が可能である．しかし，複数例外はすべて前述のように取り扱われるため，プログラマに複数発生した例外をそれぞれ個別に取り扱うことや，1 つでも例外が発生した時点でただちに例外処理を行うという選択肢はない．

5.4 Juice との比較・検討

Juice 言語では，例外コンテキストを **complain** ブロック文によって定義する．これによって，例外コンテキストの最小の単位がメソッドである AspectJ とは異なり，さらに小さな領域を例外コンテキストとして表現できるのでより柔軟な例外処理の記述が可能である．また，Juice 言語では AspectJ と同様に例外処理を通常のアプリケーションプログラムから分離して記述することができる．このように例外処理戦略を例外コンテキストとは別ファイルに定義する場合，それらを対応付ける手段が必要である．たとえば，1 つのクラス内の個々のメソッドはそのメソッドのシグネチャによって識別することができる．しかし，同一メソッド内に複数存在する単純なブロック構文は，新たに特別な仕組みを組み込まない限り一意に識別することができない．Juice 言語では，アプリケーションプログラムとは別のファイルから個々の **complain** ブロック文を識別するのではなく，例外処理戦略として用いる例外ハンドラクラスをあらかじめアプリケーションプログラム側で指定することで任意のブロックとの対応付けを実現している．

また，それぞれの並行例外処理に関して考える．ABCL/1 や Arche は，我々が提案する並行例外処理モデルのそれぞれ一方のみを提供する．それに比べて，Juice 言語はプログラマがその両方を選択的に利用することができるので，より複雑な例外処理戦略を表現することが可能である．

6. 今後の課題

Juice 言語では，オブジェクトへのメッセージの並行送信を表現する方法として，構造的な構文より記述力が勝る **async** 修飾子を提供する．このような場合に対応する例外処理機構を実現する際には，例外の伝播に関する問題を解決する必要がある．例外コンテキストは入れ子構造を形成することができ，例外はその例外コンテキストで捕捉できない場合，1 つ外の例外コンテキストへと伝播する．例外コンテキストがオブジェクト間にまたがる場合は，そのメッセージ送信の逆順に従って例外が捕捉されるまで伝播する．**async** 修飾子を使って表現するような並行処理モデルでは同期をとらない場合がある．このため，送信側のオブジェクトにおいて，メッセージ送信時とそのメッセージに起因した例外が伝播されて来たときとでは例外コンテキストが異なる可能性がある．このように非同期メッセージ送信においては，例外コンテキストを構造的な構文で表現することが困難である．ABCL/1 では，complaint destination を利用して例外の伝播先を指定することでこれを解決している．したがって，Juice 言語で非同期例外を取り扱うためには，ABCL/1 で用いられているような非構造的な例外処理記述モデルを提案する必要がある．

7. ま と め

本論文では，並行処理中の例外処理に適した並行例外処理モデルと，例外処理記述にともなうプログラムの可読性の低下に関する問題を解決する例外処理構文を提案した．特に，並行メッセージ送信構文 **par** ブロック文による並行処理中に発生する例外に注目し，プログラマが言語からこのときに発生する複数の例外に対しても取り扱うことが可能となった．我々の提案した並行例外処理モデルを用いれば，例外が発生した場合ただちに例外処理へ移行するというプログラム全体の実行効率を優先するモデルと，ブロックで同期をとり例外処理へ移行するモデルの両モデルをプログラマが Juice 言語から選択的に利用可能である．

さらに，アプリケーションプログラムを記述したファ

イルから例外処理に関するコードを分離する方法を提案した．これによって，例外処理コードそのものの再利用が可能となった．さらに，例外処理コードを記述したファイルのみを変更するだけで，例外処理戦略のみが異なるアプリケーションを容易に開発することが可能となった．

また，1つの適用例として，本論文で提案した並行例外処理モデルおよび例外処理構文を我々が開発中の分散オブジェクト指向計算環境 Juice に導入し，その有効性を検討した．実装の結果，Java の **try-catch** 構文による例外処理に比べて実行時間が約 10 倍遅くなった．ただし，例外が発生しない場合，今回の実装した例外処理機構による実行時間への影響はほとんどない．例外処理の記述性に関しては，記述量が少なくて済む，例外処理内容の変更および修正が容易である，例外処理コードそのものの再利用が可能である，などの点で提案モデルを導入した Juice 言語の記述力の方が **try-catch** 構文による記述を行う Java より優れている．また，Juice の実装にはまだまだ考慮すべき点が多数残されており，今後さらなる実行遅延の軽減が期待できる．これらの理由から，現段階において実行遅延が 10 倍で収まったのは許容の範囲内だと考える．

参 考 文 献

1) AspectJ Web Site. http://www.aspectj.org/
2) Dijkstra, E.W.: Cooperating sequential processes, *Programming Languages*, Academic Press (1968).
3) Dony, C.: Exception Handling and Object-Oriented Programming: Towards a synthesis, *Joint Conference Ecoop-OOPSLA'90*, pp.322–330 (1990).
4) 平野 聡：*HORB Web Site*, 電子技術総合研究所 (1999). http://openlab.etl.go.jp/horb-j/
5) Ichisugi, Y. and Yonezawa, A.: Exception Handling and Real Time Features in an Object-Oriented Concurrent Language, *Lecture Notes in Computer Science*, pp.92–109 (1989).
6) Issarny, V.: An Exception Handling Model for Parallel Programming and its Verification, *ACM SIGSOFT*, pp.92–100 (1991).
7) Issarny, V.: An exception-handling mechanism for parallel object-oriented programming: Toward reusable, robust distributed software, *Journal of Object-Oriented Program*, Vol.6, No.6, pp.29–40 (1993).
8) 加藤健士，小田謙太郎，吉田隆一：適応型分散オブジェクト指向環境の実現，情報処理学会九州支部火の国情報シンポジウム 2001，pp.263–270 (2001).
9) Kiczales, G., Lamping, J., Maeda, A.M., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proc. 11th European Conference on Object-Oriented Programming*, pp.220–242 (1997).
10) Oda, K., Tazuneki, S. and Yoshida, T.: The Flying Object for an Open Distributed Environment, *Proc. 15th IEEE International Conference on Information Networking* (*ICOIN-15*), pp.87–92 (2001).
11) Sun Microsystems: *Inc. Java*(*TM*) *Language Specification* (1996).
12) Sun Microsystems: *Inc. Java*(*TM*) *Remote Method Invocation Specification* (1996).
13) Tazuneki, S. and Yoshida, T.: Concurrent Exception Handling in a Distributed Object-Oriented Computing Environment, *Proc. 7th International Conference on Parallel and Distributed Systems* (*ICPADS2000*): *Workshop*, pp.75–82 (2000).
14) 米澤明憲，柴山悦哉，Briot, J.-P.，本田康晃，高田敏弘：オブジェクト指向に基づく並列情報処理モデル ABCM/1 とその記述言語 ABCL/1，コンピュータソフトウェア，Vol.3, No.3, pp.9–23 (1986).

(平成 13 年 6 月 6 日受付)
(平成 13 年 12 月 18 日採録)

尋木 信一（学生会員）

1996 年九州工業大学情報工学部知能情報工学科卒業．1998 年同大学大学院情報工学研究科博士前期課程情報科学専攻修了．現在，同大学院博士後期課程在学中．分散オブジェクト技術，分散処理システム等に興味を持つ．

小田謙太郎

1999 年九州工業大学情報工学部知能情報工学科卒業．2001 年同大学大学院情報工学研究科博士前期課程情報科学専攻修了．現在，同大学院博士後期課程在学中．分散オブジェクト技術，適応型分散オブジェクトシステム等に興味を持つ．

吉田 隆一（正会員）

1982 年慶應義塾大学工学部電気工学科卒業．1987 年同大学大学院工学研究科博士後期課程電気工学専攻修了．工学博士．同年九州工業大学情報工学部知能情報工学科助手．1990 年同助教授．1993 年から翌年にかけてオレゴン科学技術大学院大学において客員研究員．オブジェクト指向計算，分散計算，分散処理システム，オブジェクト指向データベースに興味を持つ．日本ソフトウェア科学会，人工知能学会，IEEE，ACM 各会員．