システムソフトウェアと
オペレーティング・システム 92－2
（２００３．２．２７）

異常状態の大域的検出と要求駆動型例外処理に関する一考察

中西 恒夫 [1,2)]　山下 俊之 [3)]　北須賀 輝明 [1)]　福田 晃 [1)]

{*tun, yamashita, kitasuka, fukuda*}*@f.csce.kyushu-u.ac.jp*

[1)] 九州大学大学院システム情報科学研究院
[2)] 九州大学システム LSI 研究センター
[3)] 九州大学工学部電気情報工学科

概要

安全で頑健なシステムを構築する上で例外処理は重要であるが，その漏れのない記述は極めて難しく繁雑である．本稿では例外処理を異常状態の検出と異常状態からの回復の二局面に分けて考察する．異常状態を漏れなく検出するべく，プログラムのコード断片が満足すべき条件を宣言的に記述し，条件を逸脱したときにはすぐに例外処理を起動する例外処理方式を考える．またそのときの異常状態検出コードの自動生成について述べる．さらに異常状態からの回復処理に見られる冗長性を除くべく，要求駆動型の例外処理を考える．

Global Exception Detection and Demand Driven Exception Handling

Tsuneo Nakanishi[1,2)]　Toshiyuki Yamashita[3)]　Teruaki Kitasuka[1)]　Akira Fukuda[1)]

[1)] Graduate School of Information Science and Electrical Engineering, Kyushu Univ.
[3)] Computing and Communications Center, Kyushu Univ.
[4)] Department of Electrical Engineering and Computer Science, Faculty of Engineering, Kyushu Univ.

Abstract

Exception handling is essential to construct safe and robust systems, however, its implementation is difficult and boring. Exception handling consists of detection of abnormal situations and recovery from those situations. We consider an exception handling model such that exception handlers are invoked as soon as a program breaks invariant conditions, described in a declarative manner, to be preserved. Code generation for detecting exceptions under the exception handling model is also described. Moreover, we consider demand driven exception handling to eliminate redundant codes which are often shown in exception handlers.

1　はじめに

今日の家電製品や携帯電話などの組込み機器は，商品に付加価値を与えるべく多機能化とネットワーク化が進められており，そのソフトウェアは複雑化の一途をたどっている．一般消費者に大量に販売されるこれらの組込み機器は，欠陥品の回収に膨大なコストが費やされるため，組込み機器向けソフトウェアの信頼性保証は開発現場における最重要課題である．一方で近年は，組込み機器の商品サイクルの短期化が進んでおり，組込み機器向けソフトウェアの開発工期短縮が厳しく求められるようになっている．

実際の組込み機器向けソフトウェアでは，製品の信頼性を保証する例外処理，すなわちシステムの異常な状態を検出し正常な状態に回復するための処理が相当な部分を占めており，開発工期においても例外処理の設計・実装・テストはかなりの時間を占める．

例外処理の記述を容易化することは，組込み機器向けソフトウェアの信頼性を向上するとともに，例外処理コードの開発工期を短縮し，ひいては製品全体の開発工期を短縮する上で極めて効果的であると考えられる．ソフトウェアの多くの異常が例外処理に起因している [2]．文献 [5] では，POSIX 準拠のオペレーティングシステムですら，相当数のテストについて異常な入力に対してエラーを発生せず，システムコールやライブラリ関数の異常終了，最悪の場合はシステムごとクラッシュしていることが報告されている．

本稿では，例外処理を「異常状態の検出」と「異常状態からの回復」の二局面に分けて考える．

例外処理記述の難しさの理由としてはおおよそ以下のものが挙げられよう．

1. **面倒である:** ユーザ入力の検査，ライブラリやシステムコールの返り値の検査が煩わしい．先送りにしているうちに忘れてしまう．
2. **組合せ論的に難しい:** 異常状態の検出を網羅的に手続きとして書くのが難しい．また，異常状態によって回復の手順が異なり得るため，異常状態からの回復処理を記述するのはさらに難しい．
3. **きれいに書けない:** 異常状態の検出には複雑な条件判断を伴い，異常状態からの回復処理は，関数からの大域脱出など構造化プログラミングとは相容れない制御の流れを伴う．例外処理の記述のために，問題の本質部分の記述がぼける．

わざわざいれた例外処理にバグが混入し，かえってシステムがおかしな振舞いをすることも少なくない．

本稿は筆者らの今後の例外処理研究の方向を示すポジションペーパーである．本稿第 2 節では，C++ や Java 等のプログラミング言語に導入されている例外処理構文，すなわち try-catch 構文について述べ，その問題点について考察する．漏れなく異常状態を検出するべく，第 3 節では異常状態の大域的検出のための try-catch 構文の拡張案とそのコード生成について議論する．第 4 節では，異常状態からの回復処理に見られる冗長性を排すべく，要求駆動型の例外処理について考え，第 3 節と同様に try-catch 構文の拡張案を示す．最後に第 5 節でまとめを述べる．

2 例外処理構文

今日，プログラミングに一般的に使用されているのは C, C++, Java 等の手続き型言語であるが，最近のたいていの手続き型言語は例外処理のための構文を有している．例外処理のための構文は，大域脱出モデル（nonlocal transfer），中断モデル（termination），再始動モデル（retry），再開モデル（resumption）に分けられる [8].

たとえば，C++[7] や Java[3] には下記の擬似コードで示される，中断モデルの例外処理構文が定義されている．try 節には通常の処理を記述する．プログラマは，try 節中において明示的にシステムが異常状態にないかどうかを検査し，システムが異常状態に陥っている場合は throw 文を用いてオブジェクトを「投じる」．オブジェクトは try 節内部で呼び出される関数の中からも投じることができる．throw 文によりオブジェクトが投じられると，計算機は try 節の以降の実行を取りやめ，catch 節を前から順番に走査する．throw 文によって投じられたオブジェクトの型と一致する型の仮引数を持つ catch 節が見つかった場合，計算機はその catch 節，すなわち例外処理を実行する．Java では finally 節をつけることができ，finally 節に記述されたコードは try 節の通常時処理，または catch 節の例外処理の実行が終わったときに必ず実行される．

```
try {
  通常処理
}
catch (型 1 仮引数 1) {
  例外処理 1
}
...
catch (型 n 仮引数 n) {
  例外処理 n
}
finally {
  終了時処理
}
```

システムが異常状態にないかどうかを検査し，throw 文を正しく投じるのはプログラマの責任であ

る．しかしながら，システムが異常状態に陥っていないかを検査するコードが実行されない以上，異常状態の検出は行われない．実プログラムにおいて，try 節内の通常処理コードは，関数呼出やループを含む複雑な制御フローとなる．さらにその制御フローはプログラム開発中のコード変更によって変化する．また，異常状態の検出あるいは異常状態からの回復のコードは，プログラム本来の動作とは異なるものであり，プログラムの可読性を大きく損ねる．通常処理部に異常状態を検出するコードを漏れなく記述するのは極めて難しい作業である．

異常状態からの回復処理には，異常が生じた状態に応じて，ほぼ同じであるが少しずつ異なるコードをいくつも冗長に書くなければならないケースが少なくない．実例を示す．図 1 は X Window でピクスマップを扱うプログラムの一部である．このプログラムでは，XCreatePixmap 関数を用いてピクスマップを，XCreateGC 関数を用いてグラフィックコンテキストを生成する．これらの GUI 資源は不要になったら，それぞれ XFreePixmap 関数，XFreeGC 関数を用いて必ず解放しなければならない．何らかの異常により処理の継続が不可能になった場合も，例外処理コード中でこれらの GUI 資源を確実に解放しなければならないが，図 1 にはそのための処理が try 節にも catch 節にも冗長に記述されている．catch 節内部のコードについては，図 2 のように，if 文を用いて記述すれば若干の冗長性を除けるものの，資源解放に関する一連の処理が追いにくくなるためプログラムの可読性の点では好ましくない．

基本的に try-catch 構文によって支援がなされる異常状態からの回復処理は，ローカル変数が配置されるスタックメモリの回復のみである．したがって，スタックメモリ外の資源，たとえばヒープメモリ上に動的に確保されるオブジェクトやファイル，ソケット，プロセス，スレッドのようにプログラム外部に置かれる資源の解放や現状回復に関する処理は，このような冗長性が生じがちである．

```
try {
  pixmap
    = XCreatePixmap (display, root,
                     width, height, depth);
  if (pixmap == None)
    throw 0;
  gc = XCreateGC (display, pixmap, 0, 0);
  if (gc == None)
    throw 1;
  ...
  /* この間でエラーが生じたら throw 2 */
  ...
  XFreePixmap (display, pixmap);
  XFreeGC (display, gc);
}
catch (int n) {
  switch (n) {
  case 0:
    break;
  case 1:
    XFreeGC (display, gc);
    break;
  case 2:
    XFreePixmap (display, pixmap);
    XFreeGC (display, gc);
    break;
  }
}
```

図 1: 冗長な例外処理の例 (1)

3 異常状態の大域的検出

第 2 節に述べた，C++ や Java による try-catch による例外処理構文では，異常状態の検出はプログラマが try 節中で明示的に漏れなく行わなければならない．異常状態検出のためのコードを確実に評価されるように漏れなく埋め込むのは困難な問題である上に，プログラム本来の動作とは異なる異常状態検出のためのコードは，プログラムの可読性を損ねる．

本節では，プログラム中のあるコード断片において常に満足すべき条件を記述し，条件を逸脱したときにはプログラマによって事前に指定されている例外処理を起動する，例外処理方式を検討する．

```
catch (int n) {
  if (n == 2)
    XFreePixmap (display, pixmap);
  if (n == 1 || n == 2)
    XFreeGC (display, gc);
}
```

図 2: 冗長な例外処理の例 (2)

3.1 例外処理の記述

例外処理コードについては，例えば C++，Java における try-catch 構文を拡張した，下記のような構文で記述するものとする．

try 節の通常コード実行途中に，i 番目 ($1 \leq i \leq n$) の unsatisfying 節に指定される条件式の不成立を検出すると，計算機は try 節を脱出して i 番目の unsatisfying 節の例外処理コードに制御を移す．また，try 節の通常コードを実行する直前に unsatisfying pre 節に指定される事前条件式を満たしているかどうか検査し，満たしていない場合は try 節の通常コードは実行せず，unsatisfying pre 節内部の例外処理コードに制御を移す．さらに，try 節の通常コードを脱出することなく完了した直後，finally 節を実行する直前には，unsatisfying post 節に指定される事後条件式を満たしているかどうか検査し，満たしていない場合は unsatisfying post 節の例外処理コードを実行する．いずれの場合も try 節を脱出する場合は，スタックは適切に巻き戻されるものとする．

```
try {
  ...
  /* 通常の処理 */
  ...
}
unsatisfying pre (事前条件式) {
  /* try 節突入時に事前条件式不成立のときの
     例外処理 */
}
unsatisfying (条件式 1) {
  /* try 節内で条件式 1 の不成立を検出したときの
     例外処理 */
}
...
unsatisfying (条件式 n) {
  /* try 節内で条件式 n の不成立を検出したときの
     例外処理 */
}
unsatisfying post (事後条件式) {
  /* try 節突入時に事後条件式不成立のときの
     例外処理 */
}
```

unsatisfying，unsatisfying pre，unsatisfying post の各節の条件式には，副作用を伴わない任意の式が記述できるものとする．また，条件式にはユーザ定義関数の呼出も含んでよいものとする．但し，それらの関数の内部における変数の更新はそれらの関数のローカル変数に対してのみ許し，その他の変数の更新はコンパイル時にエラーとして扱う．これは条件式評価中にシステムの状態を変更されることで，プログラム中に発見されにくバグが潜むことを防ぐためである．さらに，副作用を伴う関数が条件式に含まれていると，条件式の等価変換に制約が加わって条件式評価コードの最適化が困難になるため，コンパイラによるコード最適化の観点からも上述の扱いは好ましい．

条件式に記述できる条件演算子が，現在の C，C++，Java 等のプログラミング言語に定義されているもので実用上十分かどうかは議論の余地が多いにあり，本研究の重要なテーマのひとつである．プログラムが満たすべき制約条件の記述については，表明言語，すなわちデバッグのためにプログラム中に埋め込む表明（assertion）を記述するための言語の研究において多くの蓄積がある．今後，これまでの表明言語の成果を調査し，実用的なものを導入していきたい．

3.2 異常状態検出コードの生成

unsatisfying pre 節と unsatisfying post 節の条件式は，それぞれ try 節実行開始前後に一度評価されるのみであるが，unsatisfying 節の条件式はプログラムの状態変化，すなわち変数の更新があるたびに評価する必要があり，実行時オーバーヘッドの原因となる．条件式評価のオーバーヘッドを軽減する方策としては以下のものが考えられる．

- 変数の更新時にはその変数が関係する条件式のみを評価するようなコードを生成する．すなわ

ち，$f(x_1, x_2, \ldots, x_m)$ の形式の条件式は，変数 x_i（$1 \leq i \leq m$）が更新されたときのみ検査するようにする．

- 複数の条件式を評価する場合は，その共通部分式の評価結果を再利用する．たとえば，ふたつの条件式 $f_1(x_1, x_2, \ldots, x_m, g(x_1, x_2, \ldots, x_m))$，$f_2(x_1, x_2, \ldots, x_m, g(x_1, x_2, \ldots, x_m))$ を評価する場合，f_1 評価時に得られた $g(x_1, x_2, \ldots, x_m)$ の結果を f_2 の評価時に再利用する．
- 条件式の各項の評価順序を最適化し，より少ない計算時間で真偽が判定されるようにする．例えば，$X_1 \wedge X_2 \wedge \ldots \wedge X_m$ の形式の条件式については，より偽になりやすく，より計算コストの小さい項から順に評価し，$X_1 \vee X_2 \vee \ldots \vee X_m$ の形式の条件式については，より真になりやすく，より計算コストの小さい項から順に評価をする．

また，条件式評価のためのコードのサイズを削減するためには，条件式に含まれる各項の評価の総数を削減する必要がある．

本研究では，二分決定木ベースのアルゴリズムを導入することを検討している [4]．二分決定木は任意の論理関数 $f(x_1, x_2, \ldots, x_m)$ を表現する高さ m の二分木である．深さ h（$0 \leq h \leq m-1$）の全ての節点はいずれかの入力変数 x_i（$1 \leq i \leq m$）に対応している．各節点の 2 本の枝にはそれぞれ，その節点に対応する入力変数の真偽に対応する，真と偽のラベルが付与されている．全ての入力変数 x_i（$1 \leq i \leq m$）に真偽値を与えると根から葉に至るパスが定まるが，葉にはそのときの $f(x_1, x_2, \ldots, x_m)$ の値，すなわち 0 か 1 の値が与えられる．二分決定木のサイズは $O(2^m)$ であるが，一般に二分決定木は図 3 の例のように簡約化される．図 3(a) は論理関数 $f(a, b, c) = (a \wedge b) \vee c$ の二分決定木，図 3(b) は簡約化された等価な二分決定木である．簡約化された二分決定木のサイズは各深さの節点にどの入力変数を割り当てるかに依存することが知られており，サイズを可能な限り小さくする多くの簡約化アルゴリズムがこれまでに開発されている．図 3(c) の二分決定木は，入力変数の割当を変更することにより，図 3(b) の二分決定木よりもさらに簡約化されている．本節で述べた例外処理方式において異常状態検出コードを生成する際は，条件式を二分決定木で表現して簡約化を施す．これは，条件式中の冗長な項の評価を削減し，また異常状態検出コードのサイズを削減することに相当する．

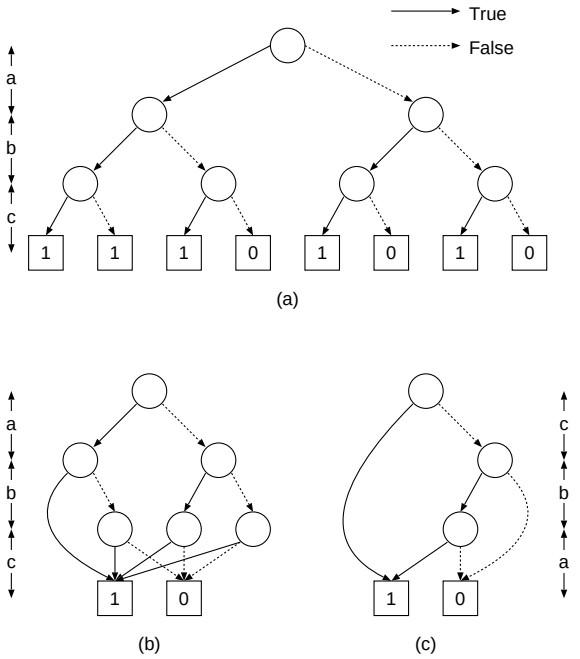

図 3: 二分決定木の簡約化

4 要求駆動型の例外処理

異常状態の大域的検出によって異常状態検出のためのコードを書くのが容易になるものの，第 2 節で述べた例外処理コードの冗長性の問題に対処できるわけではない．例外処理コードに見られる冗長性の問題に対処するべく，本節では要求駆動型の例外処理構文の案を考える．

異常状態からの回復処理が複数項目の処理からなる場合，異常が生じた状況によって処理項目に違いがあるため，例外処理コードには冗長性が生じるものと考えられる．各処理項目の実行順序には依存関係があり，異常状態から正しく回復するにはこの依

存関係の定める順序に沿って，必要な処理項目を実行しなければならない．

以上の考察に基づいて try-catch 構文を下記のように拡張する．catch 節，ならびに前節で述べた unsatisfying 節にはラベルと依存元ラベルのリストが付与できる．catch 節あるいは unsatisfying 節の実行が完了すると，その節のラベルが依存元ラベルのリストに記述されている catch 節または unsatisfying 節があれば，その節に制御を移す．この実行モデルは分割コンパイル支援ツール make に見られるものと同様である．

```
try {
  ...
  /* 通常の処理 */
  ...
}
exception ラベル 1: catch (型 1 仮引数 1) {
  例外処理 1
}
...
exception ラベル n: catch (型 n 仮引数 n)
| 依存元ラベル 1 | ... | 依存元ラベル m {
  例外処理 n
}
```

図 4 は，第 2 節で述べた X Window のピクスマップ処理における例外処理を，上述の構文を用いて記述したものである．図 1 の例ように異常状態からの回復処理に同じコードの再出はなく，図 2 の例とはちがって回復処理の各項目間の依存関係もわかりやすくなっている．

例外処理コードにおける冗長性を排するには，オブジェクト指向言語における継承の考え方を導入することも効果的に思われ，実際に例外処理の継承が可能なモデルも存在する [1]．今後，例外処理を階層化することで catch 節の継承を可能とし，要求駆動型の例外処理においてさらに例外処理コードの記述量の削減を図りたい．

5 まとめ

例外処理は安全で頑健なシステムを作る上で極めて重要であるが，プログラムの本質の動作とは異なるものであるためその記述は煩わしい．また大規模なシステムでは漏れなく記述することは難しい．本稿では，例外処理を異常状態の検出と異常状態からの回復の二局面に分け，異常状態の大域的検出，ならびに要求駆動型の例外処理について考察した．

```
try {
  pixmap
    = XCreatePixmap (display, root,
                     width, height, depth);
  if (pixmap == None)
    throw Exception0 ();
  gc = XCreateGC (display, pixmap, 0, 0);
  if (gc == None)
    throw Exception1 ();
  ...
  /* この間でエラーが生じたら
     throw Exception2 () */
  ...
  XFreePixmap (display, pixmap);
  XFreeGC (display, gc);
}
exception catch (Exception0 e) | EXFreeGC {
}
exception EXFreeGC: catch (Exception1 e)
| EXFreePixmap {
  XFreeGC (display, gc);
}
exception EXFreePixmap: catch (Exception2 e) {
  XFreePixmap (display, pixmap);
}
```

図 4: 要求駆動型例外処理の例

プログラム中のあるコード断片において常に満足すべき条件を宣言的に記述し，条件を逸脱したときにはプログラマによって事前に指定されている例外処理を起動する例外処理方式を考え，try-catch 構文を拡張した．本稿では，このように異常状態を大域的に検出できるようにすることによって，漏れのない異常状態の検出を図った．また，プログラム中のコード断片が常に満足すべき条件を二分決定木で表現し，それを簡約化することにより，異常状態検出コードを最適化することを述べた．

異常状態からの回復処理は，異常が生じた状況によって処理項目に違いがあるため，例外処理コードには冗長性が生じがちである．本稿では，異常状態からの回復処理を，処理項目ごとに分け各処理項目

間の依存関係を明示的に記述するよう try-catch 構文を拡張することで，例外処理コードの冗長性を除くことを図った．異常状態からの回復処理時には，明示的に記述された依存関係に従って，処理項目を要求駆動的に処理する．

本稿は，関数内部のミクロ的な例外処理しか議論しておらず，内容も例外処理のモデルとそれに則した構文の提示というプログラミング言語面にとどまっているが，今日の例外処理研究はさまざまの分野を跨ったものとなっている [6]．今後は，形式仕様記述，テスト，プログラミング言語，コンパイラ，プロセッサアーキテクチャの軸から例外処理を縦断的に研究していきたい．

謝辞

本研究は，平成 14 年度科学研究費補助金若手研究 B（課題番号: 14780218）の助成を受けている．

参考文献

[1] P. A. Buhr and W. Y. Russel Mok, “Advanced Exception Handling Mechanisms,” *IEEE Trans. on Software Engineering*, Vol.26, No.9, pp.820–836, Sep. 2000.

[2] F. Cristian, “Exception Handling and Tolerance of Software Faults,” *Software Fault Tolerance,* pp.81–107, Wiley, 1995.

[3] J. Gosling, B. Joy, and G. Steele, *Ths Java Language Specification*, Addison-Wesley, 1997.

[4] 石浦菜岐佐, 「BDD とは」, 情報処理, Vol.34, No.5, pp.585–592, 1993 年 5 月.

[5] P. Koopman and J. DeVale, “The Exception Handling Effectiveness of POSIX Operating Systems,” *IEEE Trans. on Software Engineering,* Vol.26, No.9, Sep. 2000.

[6] D. E. Perry, A. Romonvsky, and A. Tripathi, “Current Trends in Exception Handling,” *IEEE Trans. on Software Engineering*, Vol.26, No.9, pp.817–819, Sep. 2000.

[7] B. Stroustrup, *The C++ Programming Language 2nd ed.*, Addison-Wesley, 1991.

[8] S. Yemini and D. M. Berry, “A Modular Verifiable Exception Handling Mechanism,” *ACM Trans. Programming Languages and Systems*, Vol.7, No.2, pp.214–243, Apr. 1985.