

5章: プログラムの単純化

ANDREAS ZELLER 著 中田秀基 監訳

デバッグの理論と実践

-なぜプログラムはうまく動かないのか-

学籍番号 60206299

氏名 吉富 楓雅

TRAFFICの手順を参考に(再掲)

- ・問題の記録(T)
- ・障害の再現(R)
- ・テストケースの自動化と単純化(AUTOMATE)
- ・感染の起源の検出(F)
- ・最も感染起源の可能性が高い箇所に着目する(F)
- ・感染の起源を特定する(I)
- ・欠陥の修正(C)



自動デバッグテクニック(再掲)

今後の章の紹介



- ・入力の単純化
- ・プログラムのスライス
- ・状態の観察
- ・状態の監視
- ・アサーション
- ・異常
- ・原因と結果の連鎖
- ・ミスから学ぶ

目次

1. 問題の単純化
2. 自動的な単純化
3. 単純化アルゴリズム
4. ddminアルゴリズム
5. まとめ

今回は本書のメイン
テーマの1つ
「差分デバッグ」
についてです

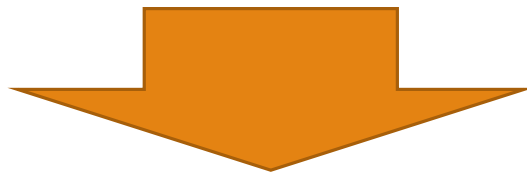
問題の単純化



pixta.jp - 18299776

デバッグでの整理

- ・問題は10,000行の入力全体に依存しているのか？
- ・ユーザとのやり取りをすべて再生する必要があるのか？
- ・一連のイベントがこの順番で起きた時にだけ障害が発生するのか？
- ・長大なメソッド呼び出しシーケンス全てが本当に必要なのか？



この整理は**単純化 (simplifying)**と呼ばれる

単純化を行う理由

1. 単純化したテストケースはコミュニケーションを容易にする
2. 単純化したテストケースはデバッグを容易にする
3. テストケースを単純化すると重複した障害報告を特定することができる

KernighanとPikeによる単純化

1

- 二分探索を行う
- 入力の半分を捨て出力がまだ不正かを確認する

2

- もし不正でなければ1つ前の状態に戻り, 残り半分の入力を捨てる

3

- 入力の単純化を続ける

自動化の欠点

単純化を手動化を行うと....

- ・単純化は単純な作業である
- ・単純化は退屈な作業である



自動化すると楽！！

自動的な単純化



自動単純化戦略の設計

1. 最初に入力の半分を除外し, テストが失敗を返すかどうかを確認する。もし失敗を返せば続けて残りの半分を処理する
2. 失敗が返されなければ前回の状態に戻り, 入力のもう半分を除外する

しかしこれではまだ不十分...

- 1 <SELECT NAME="priority" MULTIPLE SIZE=7> (40文字) ✕
- 2 <SELECT NAME="priority" MULTIPLE SIZE=7> (20文字) ○
- 3 <SELECT NAME="priority" MULTIPLE SIZE=7> (20文字) ○

自動単純化戦略の設計

3. 二文探索では不十分な場合, さらに小さなパーツに分解する

＜4分の1分割の場合＞

- 1 <SELECT NAME="priority" MULTIPLE SIZE=7> (30文字) ○
- 2 <SELECT NAME="priority" MULTIPLE SIZE=7> (30文字) ×
- 3 <SELECT NAME="priority" MULTIPLE SIZE=7> (20文字) ×
- 4 <SELECT NAME="priority" MULTIPLE SIZE=7> (10文字) ○

単純化アルゴリズム



pixta.jp - 23881028

単純化アルゴリズム

自動化戦略の定式化を実現する一般的なアルゴリズム

仮定: c を入力とする, テスト関数 $\text{test}(c)$ を用意

障害発生: ($\times$, 「失敗」)

テスト成功: ($\bigcirc$, 「成功」)

それ以外の場合: ($?$, 「不定」)

単純化アルゴリズム

障害を誘発する入力 C_x が部分集合(前半 C_1 および後半 C_2)に分割可能であるとする以下3つのいずれかが発生

- ・ C_1 (前半)を除外したものが失敗した場合:

$\text{test}(C_x / C_1) = \times$ の場合は $C'_x = C_x / C_1$ として分割を進める

- ・ C_2 (後半)を除外したものが失敗した場合:

$\text{test}(C_x / C_2) = \times$ の場合は $C'_x = C_x / C_2$ として分割を進める

- ・ 粒度を高める:

いずれでもない場合は分割の粒度を高める

単純化アルゴリズム(一般化)

C_x を C_1 から C_n の部分に分割する場合

いずれかの部分集合 (C_i) を除外したものが失敗:

ある $i \in \{1, \dots, n\}$ に対して, $\text{test}(C_x / C_i) = \times$ が成り立つ場合,
 $C'_x = C_x / C_i$, $n' = \max(n - 1, 2)$ として分割とテストを続ける

粒度を高める:

差集合が成功した場合 $C'_x = C_x$, $n' = 2n$ として分割とテストを続ける. C_x がこれ以上分割できなければ終了.

ddminアルゴリズム



pixta.jp - 31548398

さらなる一般化

プログラムの実行に影響を与える条件

入力データ ...の他に

時間 コミュニケーション スレッドスケジューリング

次の目的は障害が起こる最小の条件を求めること

ZellerとHildebrandt(2002)提案:

ddminアルゴリズム

ddminアルゴリズムの概要

- ・プログラムの実行方法はある設定(C)によって決定
 - ・関数 $\text{test}: 2^C \rightarrow \{\times, \circ, ?\}$ という関数を用意. この関数は, 与えられた設定 $c \in C$ を調査し, 成功(期待した障害を発生させる: $\times$), 失敗(期待した障害を発生させない: $\circ$), あるいは不定(?)という結果を返す
 - ・ $\text{test}(C_x) = \times$ が成り立つような設定 $C_x \in C$ を障害を発生させる設定とする. 条件が存在しない設定 (つまり $\text{test}(\emptyset) = \circ$) の場合はテストをパス.
 - ・差分デバッグのアルゴリズム $\text{ddmin}(C_x)$ によって障害を誘発する設定 C_x を最小化. 最小化された設定 C'_x は $C'_x = \text{ddmin}(C_x)$ で表される. このとき $C'_x \in C_x$ および $\text{test}(C'_x) = \times$ が成り立つ.
- また設定 C'_x は最小であるため, この設定に含まれるすべての条件が障害と関連.

ddminアルゴリズム

ddminアルゴリズムは, 下の $\text{ddmin}'(C'_x, n)$ を用いて

$\text{ddmin}(C_x) = \text{ddmin}'(C'_x, 2)$ と定義される.

$\text{ddmin}'(C'_x, n)$

C'_x

$|C'_x| = 1$ の場合

$= \text{ddmin}'(C'_x / c_i, \max(n - 1, 2))$

$\exists i \in \{1 \dots n\} * \text{test}'(C'_x / c_i) = \times$ の場合

$\text{ddmin}'(C'_x, \min(2n, |C'_x|))$

$n < |C'_x|$ の場合

C'_x

それ以外

まとめ



まとめ

- ・様々な観点から単純化は必要不可欠
- ・単純で退屈な手作業を自動化するためのテクニック
- ・単純化アルゴリズムの一般化を突き詰めたのがddminアルゴリズム