プログラム動作の類似度評価に基づく探索型デバッグ手法の拡張

植木克彦、岡本渉、田村文隆、平山雅之

(株) 東芝　研究開発センター　システム技術ラボラトリー
{ueki, okamoto, tamu, hirayama}@sel.rdc.toshiba.co.jp

概要

プログラムの動作の類似度評価に基づく探索型デバッグ手法を、動作比較からデータ比較に拡張する方法について提案する。探索型デバッグ手法は、ソフトウェアのデバッグを効率的に行うための手法で、デバッグ時に、テスト時の動作記録（ログ）を比較し、不具合の原因となっている可能性の高い箇所を探し出し、この情報に基づいて不具合の原因箇所を容易に特定する。本報告では、比較対象として扱うログの種類を動作からデータ値に拡張する方法を提案する。データ比較を行う上で注意すべき点は、データ値の違いを動作の違いと同等に扱わないことと、データ値のバリエーションをそのまま比較に用いないことである。この２点に注意を払いデータ比較をおこなうことで、探索型デバッグ手法の精度をより向上させることができると考えている。

1. はじめに

近年、ソフトウェアの機能・規模は増加の一途をたどっており、これに伴いテスト、デバッグといった作業も困難を極めている．本稿では、ソフトウェアのデバッグを効率的に行うための手法として、デバッグ時に、テスト時の動作記録を比較して不具合の原因となっている可能性の高い箇所を探し出し、この情報に基づいて不具合の原因追求を進めてデバッグを効率的に行う手法：探索型デバッグ手法を提案する。

デバッグ時には通常、デバッガなどのツールを使い動作を確認しながらソースコードを調査するが、ソースコード全体を見直したりはしない。それどころか、プログラム実行開始から異常動作発生までの流れ全てを追う事すら困難な場合も多く、疑わしい動作やそれに関係するソースコードを選択し、集中的に調査する事になる。

このようなデバッグ手法においては、プログラムの動作を分析・把握して疑わしい動作やソースコードを選び出すスキルが重要であり、これよってデバッグ効率に大きな差が生ずる。

我々が提案している探索型デバッグ手法は、プログラム実行時に記録した動作ログを分析し、疑わしい動作と優先的に調査対象とすべきソースコードを提示するための手法で、デバッグ効率を向上させる事を目的としており、概ね正常に動作する中で稀に発生するバグを主なデバッグ対象としている。ソフトウェア開発のライフサイクルにおける単体試験後期

Katsuhiko UEKI, Wataru OKAMOTO, Fumitaka TAMURA, Masayuki HIRAYAMA, Research and Development Center, Toshiba Corp.

からシステム試験までのデバッグで利用することを想定している。

本手法によるメリットとして、疑わしい領域を絞り込むことでプログラム全体の動作を理解しなくてもデバッグ作業を進められる点が挙げられる。この点は、特に大規模なソフトウェア中のバグ原因の特定に効果を発揮すると考えている。

2. 従来手法とその課題点

一般的にデバッグでは、バグが見つかった時点から徐々に動作の流れをさかのぼりながら、プログラムの内部状態を見て動作を確認してバグの原因となった処理を探りだし、得た情報をもとにして正しく動作するようソースコードを修正する。

従来から、動作をさかのぼってプログラムの内部状態を確認するために、何度もプログラムの実行を繰り返して徐々に前の状態を再現してゆくサイクリカルデバッグ手法を始め、プログラムの内部状態を記録しておき再現する逆実行手法や、単に動作ログ（トレース）を記録して調査するデバッグ手法など様々なデバッグ手法が提案・利用されている。この中には我々が提案する探索型デバッグ手法と同様、動作理解にログを利用する手法も多く見られる[1][2][3][4][5]。

これに加え、動作の流れの中でさらに調査対象個所を限定する手法として、データフローを利用して特定のデータに関係したソースコードを選びだすプログラムスライシングがある。しかし、スライシング手法を用いる場合でも調査範囲を限定するに留まり、その範囲の中で調査順序の優先度付けまではしてくれず、やはりバグが見つかった時点から徐々に流れをさかのぼりながら、順にソースコードを調査することになる。

3. 探索型デバッグ手法の概要

探索型手法の特徴は、異常動作、正常動作の違いから、バグ原因の領域を絞り込み、かつ、調査順序の優先度付けを行う点にある[6][7]。

本手法では、バグ原因領域を絞り込む方法として、異常動作と正常動作の違い（差分）部分にバグ原因があると仮定して絞り込むデバッグ手法と、異常動作中に現れる特徴的な動作にバグ原因があると仮定して優先的に調査対象とするデバッグ手法を併用する。これらはいずれも、単独でデバッグに有効な手法だが、探索型デバッグ手法ではこの 2 手法を併用することで、有効性が増大すると考えている。以下、探索型デバッグの手順とバグ原因領域を絞り込む方法について、それぞれ説明する。

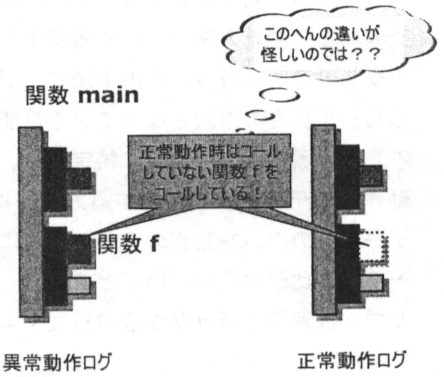

図 1：探索型デバッグ手法の基本アイデア

3.1 探索型デバッグの手順

探索型デバッグ手法のおおまかな流れは以下の通りで、多数の正常動作と異常動作の動作の違いを元にして、バグの位置を推定する。

1. 実行ログの記録
   プログラムの実行時の動作をイベントとしてログに記録する。
2. ログの分析
   正常動作・異常動作を含む多数のログを用いてイベントを分析し、異常動作とよく似た正常動作を選び出す。さらに、異常動作ログに記録されたコード実行、変数操作、入力などのイベントの中から、特にバグの原因と疑わしいイベントを探しだす。

3. ソースコードの調査
選び出した正常動作と異常動作を比較して差分を見つけるとともに、その中でより疑わしいイベントを選び、そのイベントに対応するソースコードを優先的に調査してデバッグを行う。

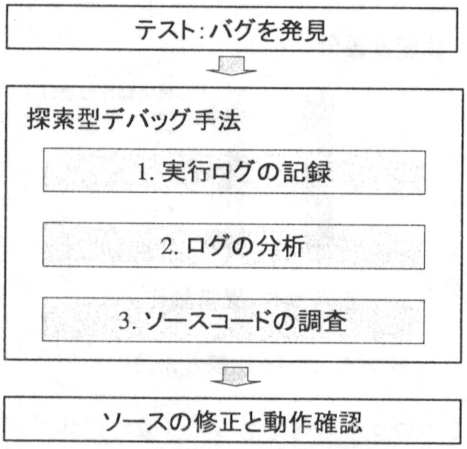

図 2：デバッグ作業の流れ

3.1.1 実行ログの記録

探索型デバッグ手法の最初のステップは、プログラムを実行しながら、実行時の動作（イベント）をログとして記録することである。

記録すべきイベントとしては、
(1) プログラムの実行制御の流れ
(2) 扱ったデータ、
という２種類が考えられる。プログラムの実行制御の流れとしてプログラム中の実行箇所を記録し、扱ったデータとして実行中に操作した値を記録する。以下、(1)の実行制御の流れに着目した解析方法を述べ、第 4 節でデータを扱うよう拡張する。

ログは、プログラムの制御が分岐する可能性のある位置(条件文やループなど) にプローブを埋め込み、実行の流れを一意に定められるレベルで記録する。

また、ログは異常動作時のものだけでなく、正常動作時のログも併せて記録する。本手法は正常動作も含めたログ同士の比較がポイントとなっている以上、各イベントの特徴を分析する母体である正常動作ログ群にどのようなログを準備するかが非常に重要である。

3.1.2 ログの分析

ユーザは、記録されたログが異常動作を示したものか、または、正常動作を示したものか、指定する。さらに異常動作ログについては、ログ中のどの時点で異常動作が観測されたか、その位置も指定する。通常のデバッグでは、この異常動作の観測位置から時間をさかのぼりながらデバッグしており、容易に特定できる。

記録されたログとユーザによる異常動作の指定に基づいて、本手法は異常動作ログとの比較に適した正常動作ログを選び出し、ユーザに提示する。選ばれた正常動作ログを「比較対象ログ」と呼ぶ。

望ましい比較対象ログとは、異常動作との差がそのまま異常の原因となっている場合である。したがって、できる限り「似ている」正常動作ログを選択することが本手法において重要なポイントとなる。ログ同士の類似性については第 3.2.1 節を参照して頂きたい。

またユーザが、比較する領域を指定することで、より効率的な比較を行うことができる。例えばユーザが既に、ある関数内(そこから呼び出されている関数も含めて) にバグ原因があると見当を付けている場合、その関数を比較範囲に指定し、関数の実行に対応したログ

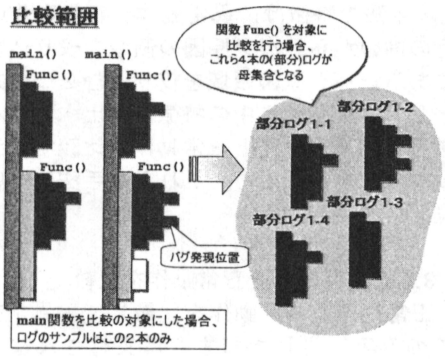

図 3：比較範囲

に限定して分析する。これにより、関係のない箇所の類似性を考慮する必要がなくなる。図３は、関数 Func()を指定しそれに対応するログを切り出してログの母集合とした例である。

この事からわかるように、ログは必ずしもプログラム実行の開始から終了までを１つの単位としない。つまり、１本のログを複数の部分ログに切り分けて、別のログとして扱える。

比較対象ログを選び出すと同時に、異常動作ログに含まれるイベントの中から、他のログになるべく含まれていない、稀なイベントを探し出す。この稀なイベントを「特徴」と呼ぶ。特徴の希少度に応じて特徴値を計算し、優先順位を設定する。

3.1.3 ソースコードの調査

比較対象ログを選択したら、それと異常動作ログとの差分情報をもとにバグ原因を探る。複数の差分箇所が存在する場合、「どの差分から調査すべきか」ということがデバッグ効率化のための重要な要素である。

本手法では、どの差分が異常動作ログにおいて特徴的(稀なイベント) であるかを示し、優先的に調査すべき位置をユーザに示す。これにより、複数の差分箇所があったとしても、より異常動作で特徴的な位置からバグ原因の調査を進めることができる。

3.2 本手法の特徴

本手法の特徴は、異常動作、正常動作の実行の違いから、バグ原因の領域を絞り込む点にある。バグ原因領域を絞り込む２つの方法として、異常動作中の特徴を検出するデバッグ手法と異常動作と正常動作を比較するデバッグ手法を併用する。これら２手法について、詳しく述べる。

3.2.1 異常動作と正常動作を比較

正常動作と異常動作を比較して、異常動作でのみ実行された（または実行されなかった）動作は、優先的に調査する価値がある。つまり、正常動作と異常動作の差分を使って、調査対象個所を限定することができる。ただし、正常動作と異常動作の違いが大きすぎると、調査対象個所の限定に役立たないため、探索型デバッグ手法では、デバッグ中の異常動作と良く「似た」正常動作を選び出して比較に用いている。

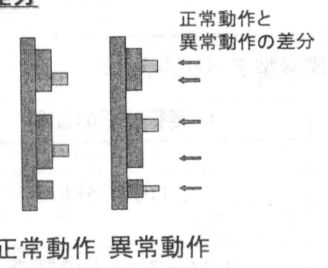

図 4: 比較と差分

２つのログの何をもって「似た」ログとみなすかは、本手法において最も重要な判断基準である。我々は、単純にログ間の差分の量(異なるイベントの数) を計るのではなく、イベントの「情報量」を重視して重み付けを行っている。イベントの出現確率をもとに計算した各イベントの特徴値を使い、ログ同士の類似性を計算するという方法を用いた[6]。

3.2.2 異常動作中の特徴を検出

異常動作中の特徴を求めるために、異常動作と正常動作の比較時にも行ったが、まずログ中の各イベントの出現確率を算出し、各ログにおける各イベントの特徴値（希少度）を求める。イベントの出現確率を計算する時には、イベントの発生順序の違いも考慮する。

異常動作中で最も特徴的なイベントを選び、イベントに対応したソースコードの調査を行う。これは、異常動作ログで発生した稀なイベントは、発生自体がバグの原因に直結している可能性が高いことに基づいている。例えば図５で、関数 subEX は２３回の実行で１回しか実行されておらず、２３回中１２回実行されているイベントｃに対応する関数

subG の実行と比べ、より優先的に調査する価値がある。

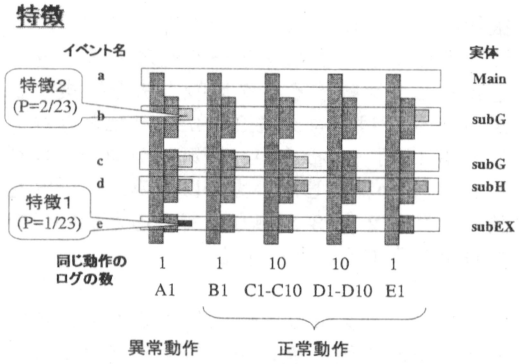

図 5 特徴

4. 探索型デバッグ手法の適用評価[7]

我々は、本手法のツール化を進めている。比較の観点としては、(1) プログラムの実行制御の流れに着目(2) データの違いに着目、という 2 種類が考えられるが、ツール化の最初のステップとして、現在のバージョンでは (1) の実行の流れに着目する手法のみ実装している。また、ログ記録機能の実装の都合上、デバッグ対象は逐次プログラムに限定している。

本手法およびツールにおいて用いたアルゴリズム[8]の有効性を評価するため、gcc 2.8.0 の cccp.c (プリプロセッサ部分) をサンプルに実験を行った。cccp.c は約 10000 行から成るプログラムである。不具合情報は gcc 2.8.1 へのバージョンアップ時の差分をもとにしている。ここでは 8 個の不具合をサンプルに用いた。実験の結果、サンプルの 8 個のバグ中、4 個については、異常動作ログにおいて最も特徴的と判断された場所そのものにバグ原因が含まれていた。さらに 3 個については、特徴的な位置の周辺を辿って行くことによって、バグ原因を発見できた。残りの 1 個は、異常動作に特徴的な位置からは、かなり離れた位置にバグ原因が存在した。

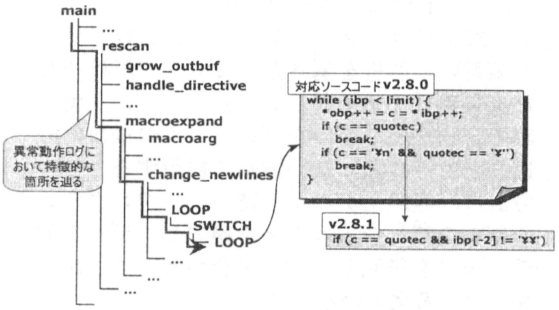

図 6：適用評価事例(gcc 2.8.0 への適用)

5. データ対応

プログラムの動作の解析では、
(1) プログラムの振舞いに基づく解析、
(2) プログラムのデータ値に着目した解析、
の 2 つの方法が考えられる。

我々が行った gcc:cccp.c と bash(未報告)を使った 2 つの実験では、プログラムの振舞いに基づく解析を行うことで、異常動作ログの中から十分良い特徴を見つけることができている。

ただし、gcc:cccp.c では、正常動作時と異常動作時で完全に同一のログが得られた例もあった。この例では、異常動作中に大きな特徴があり、そこにバグの原因があったが、デバッグ対象プログラムの種類や状況によってはプログラムのデータ値に着目した解析が必要となる可能性も考えられる。

前節まではおもにプログラムの振舞いを元にした不具合特定方式を示してきたが、本節では、更なる特定精度の向上を実現するための、データ値に着目した解析方法について、探索型デバッグ手法の流れに沿って考察する。

5.1 探索型デバッグのデータ拡張

5.1.1 データの記録

本稿では、プログラム実行中に扱ったデータ値を自由にログとして記録できる事を前提として、議論を進める。

5.1.2 データの分析

ログの解析にプログラムのデータ値を利用して解析精度を向上させる場合、プログラムの振舞いの解析を基本としてログの解析をした上でデータ値に着目した補正を行い、実行時のデータ値の違いをプログラムの振舞いの違いと対等に扱わない方針を採用する。この方針は、経験上デバッグ時には、プログラムの制御の流れを追うことを優先し、データのチェックを後回しにする場合が多いことから、妥当だと考えている。

例えば、図 7 の関数 functionA が引数 a が 0, 1, 2 と 3 回呼ばれた場合に、関数 functoinA に関するログは図 8 のように記録された場合、ログ 1 と 2・ログ 1 と 3 はそれぞれ振舞いが異なっていると考えるが、ログ 2 と 3 は同一と見なせる。

```
1  void functionA(int a){
2    if (a==0) {
3      functionB(a);
4    } else {
5      functionC(a);
6    }
7  }
```

図 7：サンプルプログラム(1)

```
ログ 1
 2  "if (a==0) {",    a=0
 3  "functionB(a)",   a=0

ログ 2
 2  "if (a==0) {",    a=1
 4  "functionC(a)",   a=1

ログ 3
 2  "if (a==0) {",    a=2
 4  "functionC(a)",   a=2
```

図 8：ログの例(1)

データの違いを意識する場合でも、振舞いの違いであるログ 1 と 2 の違いに比べると、ログ 2 と 3 の違いは些細な違いとデバッグ時には考えたい。

この制約を守るために、プログラムの振舞いの違いを優先した上で、特にプログラムの振舞いが完全に一致している部分に関して、関数呼び出しの引数やデータ処理の値の違いを識別して比較するログ比較方法を提案する。図 8 の例では、ログ 1,2,3 があったときログ 2 と 3 は同一と扱うが、振舞いが同一のログ 2 と 3 しか存在しない場合にのみ、データを調べてログ 2 と 3 が異なっていると扱い、特徴値を計算することにする。

また、データ値の比較に基づく不具合特定では、単純に値の違いを比較するだけでは、値が完全に一致する場合と多数の違う値が見つかる場合が発生して、比較が意味をなさない可能性が高い。

例えば、文字列を引数に持つ関数 functionA(char* name) が "one", "two", "three", "four", "and five" を引数にして 5 回呼ばれる場合に、単純に値の違いを比較すれば 5 回の違う動作があったとして特徴値を計算してしまい、それぞれ大きな特徴値を持つことになる。しかし、通常のデバッグ時には、name にスペースが含まれているという理由で"and five"の動作に着目して分析するが、その他のログは特に気に留めず後回しにする場合が多い。

これを反映する手段として、値を分析して特殊な値を分類する方法が有効である。この例では、含まれる token の数で分類すると、1 つと 2 つに分類され、"and five"の場合の動作が特徴として現れる。値を分析する方法としては、値域をユーザが指定する方法のほかに、境界値分析、統計処理など、データ解析の様々な手法が利用できる。

5.1.3 ソースコードの調査

データの違いを反映した特徴値計算によって比較対象ログが選択されたら、それと異常動作ログとの差分情報をもとにバグ原因を探る。ただし、最初はデータの違いを使わず動作の違いだけを用いて計算した特徴値を使い

デバッグし、良さそうな特徴が得られなかった場合にのみデータの違いを使うのが良いと考える。

5.2 データの比較が有効な例

関数呼び出しの履歴をログとして記録した場合のログの例を図 9 に示す。いずれのログも関数 f, g, h, k を呼んでいるのだが、順番、数ともに一致しており、比較してもなんの特徴も見つけることができない。

```
ログ 1:
  f g h g h g h g h   k g h g h g h g h
ログ 2:
  f g h g h g h g h   k g h g h g h g h
ログ 3:
  f g h g h g h g h   k g h g h g h g h
```

図 9：ログの例(2)

これに関数呼び出し時の引数を、ログとして記録した場合が図 10 である。値のバリエーションが多く、単に比較すると特徴だらけになってしまう。

```
ログ 1:
  f(0)   g(0)   h(180) g(1) h(70)
  g(2)   h(0)   g(3)   h(90)
  k(0)   g(0)   h(180) g(1)   h(250)
  g(2)   h(250) g(3)   h(340)
ログ 2:
  f(4)   g(4)   h(120) g(5)   h(30)
  g(6)   h(100) g(7)   h(30)
  k(4)   g(4)   h(120) g(5)   h(150)
  g(6)   h(250) g(7)   h(280)
ログ 3:
  f(8)   g(8)   h(80)   g(9)   h(130)
  g(10) h(40)   g(11) h(-10)
  k(8)   g(8)   h(80)   g(9)   h(210)
 g(10) h(250) g(11) h(250)
```

図 10：ログの例(3)

ここで、引数の値を負の値、0、正の値の3つに分類し、それぞれ"-", "0","+"に置き換えると、図 11 のようになる。バリエーションが減り、見通しが良くなっている。

```
ログ 1:
  f0 g0 h+ g+ h+ g+ h0 g+ h+
  k0 g0 h+ g+ h+ g+ h+ g+ h+
ログ 2:
  f+ g+ h+ g+ h+ g+ h+ g+ h+
  k+ g+ h+ g+ h+ g+ h+ g+ h+
ログ 3:
  f+ g+ h+ g+ h+ g+ h+ g+ h-
  k+ g+ h+ g+ h+ g+ h+ g+ h+
```

図 11：データ処理後のログの例

最後に、ログ3で異常動作が発生していた場合には、ログ3に最も似たログとしてログ2が選ばれると同時に、ログ3の特徴として 9 回目の関数呼び出しでの h-が選び出される。これで漸く、デバッグに有効な情報を得ることができた。

6. おわりに

探索型デバッグのデータ比較への拡張方法について紹介した。本手法により、動作比較では良い特徴を見つけられない場合でも、データ比較を行うことにより特徴を見つけられる可能性が出てくることを示した。

データ比較手法は、プログラム実行の結果誤った値が出力されるタイプのバグに有効に働くと考えている。これまでデバッグ実験をした範囲では、プログラムの動作の違いを分析するだけで有効な特徴を検出できているが、これはデータ処理を多く含むプログラムを対象にしていないことが原因だと推測している。また、デバッグ時には、データ値が表示されているほうがログとして見易く望ましいこともあり、探索型デバッグ手法のツールとして少なくともデータの記録と表示は必要性が高い。

デバッグ作業では初心者とベテランのスキル差が問題となるが、探索型デバッグ手法は、

特にデバッグに不慣れな初心者であっても有効に利用でき、デバッグ効率を向上できるデバッグ手法だと考えている。

参考文献

[1] James R. Larus, University of Wisconsin-Madison, Efficient Program Tracing, IEEE Computer, May 1993, 52-61

[2] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. ACM computing surveys, 21(4): 593-622, 1989

[3] KIJIRO ARAKI, et al. Kyushu University, A General Framework for Debugging. IEEE Software, May 1991, 14-20.

[4] BOGDAN KOREL, PELAS-Program Error-Locating Assistant System, IEEE Trans. On Software Engineering, Vol.14, No.9, SEP 1988.

[5] JONG-DEOK CHOI, IBM Thomas J. Watson Research Center, et al. Techniques for Debugging Parallel Programs with Flowback Analysis, ACM Trans. on Programming Languages and Systems, Vol.13, No.4, October 1991, Pages 491-530.

[6] 植木克彦他, "探針型デバッグ手法の提案", 信学技報, Vol.100, No.186, pp.1-8, (2000-7)

[7] 岡本渉他, "探針型デバッグ手法の実現(1) - 概要と評価", 62 回情処全国大会 2Z-2 (2001-03)

[8] 田村文隆他, "探針型デバッグ手法の実現(2) - アルゴリズム", 62 回情処全国大会 2Z-2 (2001-03)

本 PDF ファイルは 2002 年発行の「第 43 回プログラミング・シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトに、下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載し、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

https://www.ipsj.or.jp/topics/Past_reports.html

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間： 2020 年 12 月 18 日 ～ 2021 年 3 月 19 日
掲載日： 2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html