静的型付き言語で動的に生成された型が含むバグの早期発見に向けた研究

横井 駿平　千葉 滋

C#はリフレクション操作によってコンパイル時に存在しない型を実行時に生成することが可能な言語であるが，この機能を用いたプログラムはバグを含みやすく，また発生したエラーの情報からバグの原因を特定することが通常のプログラムの場合よりも困難であるため，実行時の型生成を伴うプログラムの記述は敬遠されてきた．本研究では動的な型生成を行うリフレクション計算に起因する実行時エラーについて，生成される型に型構築時の実行時フローの情報を attribute として埋め込むこと，および埋め込まれた情報をエラー発生時に参照することでエラーの原因と考えられる部分をエラーメッセージ内で示唆し，この種のバグの早期発見，早期解決を支援するための仕組みを提案する．

1　はじめに

C#はリフレクション操作によって実行時に型を生成することが可能な言語であるが，そのような操作はバグの原因となりやすく，また型生成操作を含むプログラムのバグは通常のプログラムの場合と比較してエラー発生時に出力されるエラーメッセージからエラーの原因を推測することが難しい．

実行時に型を生成する操作は通常のプログラムの中ではあまり用いられないが，処理の高速化や高機能なライブラリの実現のため，主にライブラリ開発者によって用いられている．この技術を用いているものの例として C#による Web アプリケーション開発用フレームワークである ASP.NET が挙げられる，Web アプリケーションはその性質上，アプリケーション実行時に実行すべきメソッドを動的に決定しなければならない場面が存在する．リフレクション操作を行うことでこの処理は実現可能であるが，通常のプログラムの実行速度に対してリフレクション操作は極めて低速であるため，実用アプリケーション内でリフレクション操作を多用することは推奨されない．そこで ASP.NET では，リフレクション操作によって動的にメソッド選択を行ったのち，動的コード生成によって選択されたメソッドを実行するためのコードをその場で生成することによって動的なメソッド実行と高速な動作の両立を実現している．

このように動的な型生成操作は高機能なライブラリやフレームワークを作る上で重要になるが，型生成操作に起因する実行時エラーメッセージの改善に関する議論はこれまで行われてこなかった．本研究では，型生成時の実行フロー情報を生成された型に埋め込むことにより，動的に生成された型およびそれを利用するコードに含まれるバグを早期発見するための手法を提案する．以下ではまず動的型生成操作そのものおよびそれに起因するエラーとエラーメッセージについて説明し (2 章)、続いてエラーメッセージを改善するための手法の提案およびその実装方法について説明を行う (3 章)．そして本研究と関連する研究を紹介し (4 章)，最後に今後の課題を述べる (5 章)．

2　動的型生成

動的な型生成処理とは，実行時のリフレクション計算によってコンパイル時には存在しない型を実行時に生成する手法である．以下のプログラム 1 は，int 型のフィールド x および y を持つクラス Vector2 を実

Shumpei Yokoi, Shigeru Chiba, 東京大学大学院 情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

行時に生成する C#プログラムの疑似コードである．

プログラム 1　動的な型生成の例

```
1  var typeBuilder = modBuilder.DefineType
       ("Vector2",TypeAttributes.Class,
       typeof(object));
2  var xBuilder = typeBuilder.DefineField
       ("x", typeof(int),FieldAttributes.
       Public);
3  var yBuilder = typeBuilder.DefineField
       ("y", typeof(int),FieldAttributes.
       Public);
4  var vec2 = typeBuilder.CreateType()!;
5  var v = Activator.CreateInstance(t)!;
```

プログラム 1 の 5 行目に登場する変数 v は，ソースコード内に現れないクラスである Vector2 型のインスタンスである．

これらの動的な型生成処理の内容はコンパイル時にチェックされないため，動的型生成に起因する問題 (必要なメソッドの実装忘れ等) は実行時エラーとして検出される．ところが，実行時エラーメッセージはエラーの内容そのものやエラーの直接の原因を示すものでしかないため，問題解決のために修正すべき部分を特定するための情報としては不十分な場合も多い．例として以下のプログラム 2 を挙げる．これは同名のフィールドを同じ型に 2 つ定義してしまうプログラムの疑似コードである．

プログラム 2　動的な型生成の例

```
1  var typeBuilder = modBuilder.DefineType
       ("Vector2",TypeAttributes.Class,
       typeof(object));
2  var xBuilder = typeBuilder.DefineField
       ("x", typeof(int),FieldAttributes.
       Public);
3  var yBuilder = typeBuilder.DefineField
       ("x", typeof(int),FieldAttributes.
       Public);
4  var vec2 = typeBuilder.CreateType()!;
5  var v = Activator.CreateInstance(t)!;
6  // 以下の行で例外発生
7  v.x = 100;
```

このプログラムは 1 とほぼ同じものであるが，3 行目の DefineField メソッドで指定したフィールドの名前が異なっている．このプログラムでは 2 行目と 3 行目で異なる同名のフィールド x の追加を行っている．その結果，生成した型 Vector2 へのアクセスを行う際，System.Reflection.AmbiguousMatchException 例外が発生する．この例外によるエラーメッセージはアクセスしようとしたフィールドが曖昧であったことしか示唆しないため，このエラーが同名のフィールドを 2 回定義したことにより発生したものであることを特定するのは難しい．

我々は，動的型生成を原因とするエラーが発生した際，なるべく早く原因を特定しバグを取り除くための仕組みが必要であると考えた．原因の特定を容易にするためには，エラーの発生時にプログラマが参照可能な情報を増やすことが重要である．なぜならば，十分な情報が存在すれば問題の発生原因を絞り込むことが可能となるためである．本研究では，動的に型を生成するリフレクション計算を対象とし，問題発生時に出力されるエラーメッセージの改善によるバグの原因の早期発見を目指す．

動的に型を生成するリフレクション計算に起因するエラーには大きくわけて「型を生成するコード内のエラー」と「生成されたコード内のエラー」，そして「生成された型を利用するコード内のエラー」が存在する．生成されたコード内，つまりコンパイル時には直接コードに現れない部分の実行によるエラーの存在が，動的な型生成を含むプログラムに発生しうるエラーの特徴である．この種のエラーには元のソースコード内にエラーの発生箇所と呼べる部分が明確に存在しない可能性があり，そのことが問題解決をより一層困難にしている．

2.1　実行時エラーの例

前述のように，動的な型生成に起因するエラーにはいくつかの種類がある．この節ではこれらのミスのうち典型的なものの例をサンプルコードを用いて紹介する．

プログラム 3 は，インタフェース型 IActor を実装したクラスを動的に生成する C#のプログラムを簡略

化したものである．C#の標準ライブラリを用いてインタフェース型を実装する型を動的に生成する際には SetInterfaceImplementation メソッドを呼び出す必要があるが，このコードはメソッド呼び出しを欠いているため，実行時，生成したクラスを IActor 型にキャストする部分 (行番号 12) で実行時例外が発生する．

プログラム 3　interface 実装忘れの例

```
1  public interface IActor
2  {
3    void Function();
4  }
5  ...
6  var typeBuilder = modBuilder.DefineType
       ("DynamicActor",TypeAttributes.
       Class,typeof(object));
7  var methodBuilder = typeBuilder.
       DefineMethod("Function",
       MethodAttributes.Public |
       MethodAttributes.Virtual,typeof(
       void),Type.EmptyTypes);
8  var exp = Expression.Block(
9    Expression.Call(typeof(Console).
         GetMethod(name: "WriteLine",
         types: new Type[]{typeof(string)
         })!),Expression.Constant("Hello
         !"));
10 Expression.Lambda(exp).CompileToMethod(
       methodBuilder);
11 var t = typeBuilder.CreateType()!;
12 var dy = (IActor)Activator.
       CreateInstance(t)!;
13 dy.Function();
```

プログラム 4 は，生成した型が持つフィールドの名前を誤ったためにフィールドアクセスの際に実行時エラーが発生するプログラムの疑似コードである．この例において，生成した型の持つフィールドの名前は age であるが，フィールドアクセスを行う部分ではフィールド Age にアクセスしようとしている．そのため、このプログラムを実行した場合，フィールド Age へのアクセス部分 (行番号 6) で missing-field を原因とする実行時エラーが発生する．

この例のようにプログラムが短い場合、ソースコード全体を確認することでエラーの発生原因がスペルミスであることが確認出来るが、より複雑なコードの中にこのようなエラーが含まれていた場合、エラーの情報のみから原因を特定することは困難である．

プログラム 4　フィールド名誤りの例

```
1  ...
2  var typeBuilder = modBuilder.DefineType
       ("Person",TypeAttributes.Class,
       typeof(object));
3  var age = typeBuilder.DefineField("age
       ",typeof(int),FieldAttributes.
       Public);
4  var t = typeBuilder.CreateType()!;
5  dynamic dy = Activator.CreateInstance(t
       )!;
6  dy.Age = 10;
7  Console.WriteLine($"Age: {dy.Age}");
```

2.2　型生成リフレクションに起因する実行時エラー

この節では，動的型生成を行うリフレクション計算に起因する実行時エラーをその発生タイミングで分類し列挙する．

1. **CreateType 呼び出し時，生成しようとした型に不備がある場合**
   CreateType メソッドは，型を表すオブジェクトから Type 型のインスタンスを生成するメソッドである．このメソッドの内部では型を表すオブジェクトの持つ情報が参照され，その整合性を確認した後に Type 型のインスタンスが生成される．この種のエラーの例として，実装したインタフェースが要求するメソッドを生成した型が持っていない場合や同名のフィールドを 2 つ以上定義した場合などが挙げられる．
2. **生成した型のインスタンスを既存の型にキャストしたとき，キャスト先の型との間に互換性がない場合**
   先述の例のように実装すべきインタフェースや継承すべき基底クラスが指定されていない場合や，誤った型をキャスト先に指定した場合が該当

する．InvalidCastException として検出される．

3. **MissingMethodException 等，生成した型が期待されるメンバを持っていないことによる実行時例外が発生するような場合**
   メンバの追加を忘れたケースや，スペルミスなどによって本来想定していたメンバが追加されない場合などが該当する．存在しないフィールドへのアクセスや存在しないメソッドの呼び出しとして検出される．

2.3 既存の処理系における動的型生成の問題点の扱い

以上でも繰り返し述べた通り，C#における動的な型生成処理はバグを含みやすいものであり，また発生したエラーの原因を特定するのも困難であった．それにもかかわらず現在においてもこの問題が解決されていない理由として我々が考えているものは以下の 2 点である．

1. 動的な型生成処理を行う機能は言語機能の全体から見れば大きいものではないため，言語機能を拡張することにより得られる恩恵が機能拡張に必要なコストに対して小さく，問題解決の優先度が低い
2. 動的な型生成処理に限らず一般にリフレクション計算はバグを含みやすいものであると認知されているが，それ故にリフレクション計算自体を避けてプログラミングを行うことが推奨されてきたため，問題解決のモチベーションが低い

つまるところ，問題そのものを回避することによってプログラマは問題を「解決」してきたのである．しかし，これらの問題は何らかの方法で解決すべきである．

3 実行時フロー情報の埋め込みによるエラーメッセージの改善

我々は，動的型生成を行うリフレクション計算を原因とする実行時エラーの発生時，エラーの原因となる間違いを含む可能性のある行をエラーメッセージ内で指摘することによりバグの原因の早期発見を支援するための仕組みを提案する．この仕組みでは，生成しようとした型に不備がある場合，元のソースコード内に原因と考えられる行が存在するならばその行をエラーメッセージ内で指摘する．例えば，1 つのメソッドをオーバーライドするメソッドが 2 つ以上定義されている場合，当該メソッドのオーバーライドメソッドを実行時に指定した全ての箇所をエラー原因の候補として列挙する．2.2 節で述べた実行時エラーについても同様に，それぞれ適切なエラーメッセージを表示する．

エラー発生原因の特定につながるエラーメッセージを生成するためには，型の構築時に行われたエラーの原因であると考えられる操作を参照し，その操作を行った行の情報をエラーメッセージ内に含めるべきである．そのためには，型構築の実行時に操作の履歴を何らかのかたちで記録しておく必要がある．提案手法では，型構築に関係する操作の実行履歴を実行時に適宜その型自身に埋め込むことで，エラー発生時の実行フローの追跡を可能とする．なお，エラーを発生させた型に過去に行われた操作のうちどれがエラーの原因であるかは，発生したエラーの種類によって個別に判断する (3.2 節で個別に説明する)．

提案手法では実行時エラー発生時に実行時エラー発生時に型構築に関する操作の実行履歴の分析を行う。このとき実行履歴に加えて制御フローの静的な情報も分析に使う。制御フローの静的な情報はコンパイル時に取得し、プログラム中に埋め込んでおく。この情報により、条件分岐などで実行時に選択されなかった実行パスに含まれる操作についてもエラーメッセージに含めることができる。例えば、実行時に行われた誤りを含む操作だけでなく実行時に必要な操作が行われなかった (条件分岐によってメソッドの追加が行われないケースがあるなど) ことによるバグの発生についても原因の特定につながる実行時エラーメッセージを生成することが出来る．

プログラマによるこれらのバグの原因の特定が困難である理由のひとつに，実行時エラーメッセージの情報量が不十分であるためエラーの発生原因を元のソースコードの広い範囲から探さなくてはならないことが挙げられる．提案手法の仕組みを用いることでプログラマは動的型生成を行うリフレクション計算に

起因する実行時エラーの発生時にエラーの原因を絞り込めるようになるため，バグの原因を早期に発見することが出来るようになると期待出来る．

3.1 実現方法

通常の C#における動的な型生成のリフレクション計算は，以下のような手順で行われる．

1. 型を表すオブジェクトを作成する
2. 作成したオブジェクトの持つメソッドを呼び，型の情報を修飾する．ここではオブジェクトに対して行うことの出来る操作の一部を挙げる
   - メソッドの追加
   - フィールドの追加
   - コンストラクタの追加
   - ジェネリックパラメータの設定
   - 基底クラスの設定
   - インタフェースの実装
   - オーバーライドメソッドの設定
3. 型を表すオブジェクトの CreateType メソッドを呼び出し，Type 型のインスタンスを得る
4. Activator.CreateInstance メソッドを用いて，前項で得た Type 型のインスタンスが表す型のインスタンスを生成する

我々は，生成する型を表すオブジェクトに対し型構築フローの情報を適宜追加することにより提案手法を実現する．より具体的には型を表すオブジェクトに対する AddMethod や AddField などの操作や条件分岐を行った直後に，実行時のスタックトレース情報から生成した当該部分のソースコード内における位置や操作内容の情報を SetCustomAttribute メソッドを用いて生成する型を表すオブジェクトに追加し，生成される型に attribute として型構築フローの情報を埋め込む．このようにして埋め込んだ情報を CreateType の呼び出し時やメンバアクセス時など実行時エラーの発生時に参照し、発生したエラーの内容を元に過去の操作からエラーに繋がるものを探し出すことで，エラー原因の特定につながる情報を含んだエラーメッセージを生成する．

以下のプログラム 5 は，プログラム 4 と同内容のプログラムを提案手法の拡張文法に従って書き直したものである．このプログラムもプログラム 4 と同様スペルミスが原因の実行時エラーを含んでいるが，このプログラムの実行時に出力されるエラーメッセージはフィールド age の情報を元にスペルミスの可能性を指摘する．なお，このプログラムにおいても実行時エラーは 7 行目で発生する．

プログラム 5　提案手法におけるフィールド名誤りの例

```
1  ...
2  let person = new Class[name = "Person
       "]();
3  let age = new Field[name = "age",type
       = int]();
4  let person = person.AddField(age);
5  typevar person_t := person.CreateType()
       ;
6  var dy = new person_t();
7  dy.Age = 10;
8  Console.WriteLine($"Age: {p.Age}");
```

3.2 エラーメッセージの例

この節では 2.2 節で示したエラーについて，様々なケースにおいて生成されるエラーメッセージの具体的な例を挙げる．

3.2.1 CreateType 呼び出し時，生成しようとした型に不備がある場合

- **同名メンバが重複して定義された場合**

  このエラーは CreateType 呼び出し時に発生する．CreateType の呼び出し時，生成される型を表すオブジェクトの attribute から実行時に同名メンバの定義を行った部分の情報を取り出し，エラーメッセージ内で該当箇所を列挙する．

- **基底クラスや実装したインタフェースが要求するメソッドが欠落した場合**

  このエラーは CreateType 呼び出し時に発生する．追加されたメソッドの中に要求されたメソッドと近い名前を持つものや，同名であるが型の異なるものを探す．ここで，メソッドの名前の「近さ」は編集距離によって決定する．「近い」メソッドが発見された場合，発見されたメソッドを追加した部分の情報をエラーメッセージに含める．「近

い」メソッドが発見されなかった場合にはメソッドの追加を忘れたものと判断し，CreateType の呼び出し自体をエラーの原因とする．

- **オーバーライド設定の欠落**
  このエラーは CreateType 呼び出し時に発生する．オーバーライドすべきメソッドと同名/同型のメソッドが存在するがオーバーライドが適切に設定されていない場合がこれに該当する．CreateType 呼び出しをエラー原因として指摘し、オーバーライドを設定するためのメソッドの呼び出しを追加するように提案する．
- **互換性のないメソッドのオーバーライド**
  このエラーは CreateType 呼び出し時に発生する．オーバーライド設定対象のミスを疑う．型に追加されたメソッドの中に本来オーバーライドが設定されるべきメソッドと「近い」メソッドが存在した場合，そのメソッドを追加した部分をエラー原因として指摘する．

3.2.2 生成した型のインスタンスを既存の型にキャストしたとき，キャスト先の型との間に互換性がない場合

- **互換性のない型へのキャスト**
  このエラーはキャスト演算の実行時に発生する．インタフェースの実装あるいは基底クラスへの追加を行う部分を実行時フローが通過しなかったことによるエラーを疑う．型構築の実行時フローから実行時に通らなかったパスの情報を取得し，通らなかったパスの中に本来行うべき処理が見つかった場合はそのパスを通らない原因となった条件分岐箇所をエラー原因の候補として指摘する．そうでない場合，設定忘れあるいは互換性を持たせる予定のない型への誤ったキャストであると判断し、CreateType 呼び出しおよびキャスト演算をエラー原因の候補とする．

3.2.3 MissingMethoException 等，生成した型が期待されるメンバを持っていないことによる実行時例外が発生するような場合

- **存在しないフィールドへのアクセス**
  このエラーはフィールドアクセスの実行時に発生する．アクセスしようとしたフィールドと「近い」名前を持つフィールドがその型に存在する場合，スペルミスがエラーの原因であると判断し「近い」名前のフィールドを追加した部分の情報をエラーメッセージに含める．そうでない場合は型生成処理内におけるフィールドの追加忘れであると判断し，その型を生成した CreateType メソッドの呼び出し部分の情報をエラーメッセージに含める．また，型生成の実行時フローが条件分岐を含んでいる場合，選択されなかったパスがアクセスしようとしたものと同名のフィールドを追加する可能性についてチェックを行う．選択されなかったパスがそのようなフィールドの追加操作を含んでいた場合，条件分岐部分もエラー原因の候補として指摘する．

3.3 実装方法

この節では，提案したシステムの具体的な実装案について解説する．提案したシステムは C#をベースとしたものであるが，既存の C#の文法のみではシステムの構築には不十分である．よって我々は，C#の文法を拡張することによって提案したシステムを実現することにした．現時点で拡張文法に対応したコンパイラは完成していないが，以下で示す方針に従うことで実装が可能であると考えている．

C#コンパイラのリファレンス実装である Roslyn はオープンソースソフトウェアとして開発されている．そのため，Roslyn のソースコードに変更を加えることで，C#の文法に加えて拡張文法にも対応したコンパイラを作成することが可能である．

Roslyn によるコンパイルのフローは以下のような構成になっている．

1. **Parse** プログラムの字句解析および構文解析を行う
2. **Declaration** プログラム内の宣言やインポートしたメタデータから名前付きシンボルの宣言を分析する
3. **Binder** 解析したシンボルとプログラム内の識別子を対応付ける
4. **Emit** 解析結果から IL を生成し，出力する

本システムの実装のため，拡張コンパイラはパー

ス処理の後にソースコードの変換処理を行う．この方針は現在の C#の文法においても foreach 文や IEnumerator/IEnumerable 型を戻り値として返す所謂「イテレータ構文」を用いたメソッドの実装に用いられている．元のソースコード内に存在するメソッドの追加やインタフェースの実装などの型を表すオブジェクトへの操作の後に，そのオブジェクトに対して SetCustomAttribute メソッドを実行するコードを追加する．また，型の構築に関わる部分に含まれる条件分岐に対し，条件分岐の前にそれぞれの分岐先においてオブジェクトに行われる可能性のある操作の情報を Attribute にセットする処理を挿入する．これにより，例えばあるメソッドを追加する処理を行うフローを通らなかったことにより実行時エラーが発生したとき，条件分岐部分がエラーの発生原因である可能性を指摘することが可能となる．

加えて，CreateType メソッドの呼び出し前および実行時に生成した型を用いる部分，例えばプログラム 1 の v のような実行時に生成した型のインスタンスが持つフィールドへのアクセスやこのようなインスタンスの既存の型へのキャストの前にもコードを挿入する．ここで挿入するコードは，直後に行おうとしている操作，例えばフィールドアクセスやキャスト演算などが実行可能であるかどうかを当該処理の前にあらかじめ検査するものである．このとき挿入されるコードでは，型の生成時に付加された attribute の情報を用いることで操作の有効性を確認し，操作が不正であると判断した場合は型の構築フローの情報を元にエラーの原因と考えられる操作の行番号を含んだエラーメッセージを出力する．

4 関連研究

C#における動的な型生成処理に関するエラーメッセージを改善する実用的なライブラリや処理系は我々が知る限り存在しないが，静的言語のコンパイル時エラーメッセージの改善を目的とした研究は存在する．

文献 [1] は，C++におけるテンプレートの誤用を原因としたエラーメッセージの改善を目的とした言語機構 concepts を提案したものである．C++のテンプレートは非常に強力な表現力を持っているが，テンプレートの使い方に誤りがあった場合、使い方そのもののミスを原因とするのではなくテンプレートの展開の失敗を原因とする大量のコンパイルエラーメッセージが生成されてしまうため，テンプレートを活用したプログラミングを正しく行うことのできるプログラマは限られていた．この論文で提案された concepts はテンプレートが要求する性質を型に近い形でまとめたものである．この機構の導入により，テンプレートの使い方に起因するエラーのコンパイルエラーメッセージはテンプレートの展開失敗ではなく concepts の制約違反を原因とするものとなり，テンプレートメタプログラミングを行いやすくなった．なお，この論文で提案された concepts は C++20 より C++の正式機能となった．

文献 [2] は，OCaml にマルチステージプログラミングの機能を導入することで型安全なメタプログラミングを可能とした MetaOCaml の設計および実装を示したものである．MetaOCaml では動的コード生成などのメタプログラミング操作に対して静的な型検査が行われるため．静的検査のないシステムと比較してエラー発生時のバグ修正が容易になることやエラーメッセージが結果的に読みやすくなることが期待出来るが，この研究はエラーメッセージの改善を目的としたものではないため我々の研究の目的にはそぐわない．

Typed Racket や Template Haskell など，メタプログラミングの可能な静的型付き言語の処理系はその他にも存在する．メタプログラミングを改善するための様々な研究の中には，我々の研究と同様エラーメッセージの改善を目的としたものもあると考えられる．現在，詳細を調査中である．

5 まとめと今後の課題

本研究では，C#における動的な型生成処理に起因するバグの早期発見ならびに早期のバグ修正のため，型生成処理の実行時フローの情報を生成対象の型に埋め込むことによる実行時エラーメッセージの情報の改善手法を提案した．具体的な実装方法の案はあるものの実装は未完成のため，提案手法を実現した実装を完成されることが今後の課題である．また，提案手法をより実用的なものとするため，本研究のターゲット

のうちより早期に発見することが可能であると考えられるエラーを静的に解析するための仕組みを検討している．

参考文献

[1] Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++, *SIGPLAN Not.*, Vol. 41, No. 10(2006), pp. 291–310.

[2] Kiselyov, O.: The Design and Implementation of BER MetaOCaml - System Description, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, Codish, M. and Sumii, E.(eds.), Lecture Notes in Computer Science, Vol. 8475, Springer, 2014, pp. 86–102.