Vol.2011-SE-171 No.23
2011/3/15

共有変数を用いたバインド方式の提案とソフトウェア開発への応用について

荻　原　剛　志[†1]

オブジェクト指向を用いない手続き型言語のプログラミングにおいても，機能を提供するコード自体を変更せずに，モジュール間の結合を変更しやすくする方法があれば，独立性，再利用性を高めることができる．本稿では，複数のモジュールが共通して参照する変数を共有変数と呼び，共有変数の変更に伴って，関連するモジュールの手続きを呼び出す仕組みを提案する．モジュール間の連携は動的に変更可能であり，呼び出される手続きの詳細はモジュール内部に隠蔽できる．この仕組みを coval と呼び，C 言語のライブラリとして実装した．Coval を利用したソフトウェアの構築，拡張の例を示し，手続き型プログラミングにおけるデザインパターンの利用可能性について考察する．

A Binding Mechanism Based on Variables Shared Among Modules

Takeshi Ogihara[†1]

If some mechanism that can easily change the binding among software modules is provided, it is possible to increase the independency and the reusability of modules, even using procedural programming languages which do not have object-oriented features. This paper proposes a new binding mechanism based on variables shared among modules. The binding mechanism activates functions associated with the shared variable when its value is changed. Binding among modules can be dynamically modified. The concrete information of functions can be hidden inside the modules. This mechanism, coval, is implemented as a library in C. This paper also show some examples of design patterns with coval.

†1 京都産業大学コンピュータ理工学部
Faculty of Computer Science and Engineering, Kyoto Sangyo University

1. はじめに

ソフトウェアの構成要素であるモジュールを利用するには，何らかの方法で，そのモジュールの機能（関数やメソッド）を呼び出す必要がある．C 言語をはじめとする手続き型（本稿では非オブジェクト指向の言語の意味で使う）のプログラミング言語では，このために，既存のモジュール自体に手を加えたり，モジュール同士を連携させるためのコード（グルーコード）を新たに追加することも広く行われている．しかし，このやり方には，ソフトウェアの品質の点でも，開発の効率という点でも問題が多い．

一方，オブジェクト指向言語では，拡張可能なデザインパターンに基づいた記述を行うことによって，既存のモジュールを変更することなく，ソフトウェアの構築を容易に行うことができる．しかし，組込みシステムの開発など，手続き型言語が使われ続けている分野は多く，手続き型言語の範囲内での効果的な設計，開発手法が必要とされている．

本稿では，ソフトウェアの中で複数のモジュールが共有する値に注目し，共有された値を介してモジュール間の連携を行う方法を提案する．さらに，この方法を利用したプログラミングがモジュールの独立性を高め，ソフトウェアの構築，拡張を容易にすることを示す．

以下では，モジュール間で共有される変数を共有変数と呼び，共有変数を介してモジュールを連携させる機構を coval と呼ぶ[⋆1]．また，モジュールを coval を介して連携させることを，モジュールをバインドすると言う．

coval は C 言語のライブラリとして実装した．必要なメモリ量，オーバーヘッドはわずかであり，OS の存在しない組込みシステム環境であっても動作することを確認している．

以下では，モジュール間の連携に関する既存の手法，特に coval の開発において参考とした Cocoa バインディングについて述べる．次に，coval の概要を述べ，プログラム例に基づいてソフトウェア開発への応用について考察する．

2. モジュール間の連携

オブジェクト（特に GUI 部品）の動作と，モデルとなるデータソースの変更を関連づける技術は，広くデータバインディングと呼ばれている．本研究では，そのようなデータとオブジェクトのバインディングではなく，モジュール間を連携させる仕組みとしてのバインディングに注目する．

⋆1 coval の名は化学の共有結合（covalent bond）から借りている．

2.1 ターゲット-アクションパラダイム

ターゲット-アクションパラダイムはバインディングではなく，メッセージ送信の仕組みを使って，GUI 部品をはじめとするオブジェクト間を連携させる技法である．NeXT 社の NeXTstep[1] において，GUI アプリケーションの構築に用いられた．記述言語である Objective-C の特徴を利用しており，現在でも Apple 社の Mac OS X および iOS で使用されている．

Objective-C はメッセージ名に相当するメッセージセレクタを使って，どんなオブジェクトにもメッセージを送信できる．この機能を利用して，メッセージの宛先（ターゲット）と送るべきメッセージを GUI 部品に設定し，クリックなどの操作を受けた場合に目的の動作を行わせるようにできる．ターゲットはまったく未知のオブジェクトでも構わず，結合を変更しても部品側のプログラムを書き直す必要はない．

2.2 Cocoa バインディング

Cocoa バインディング[2] は Mac OS X 10.3 で導入された，データバインディングに基づく開発手法である．複数のオブジェクトのプロパティを関連づけて値を共有させておき，値が変化した場合はそれぞれのオブジェクトのメソッドを起動することができる．

共有される値は，MVC（モデル・ビュー・コントローラ）に基づく設計の，モデルの部分を実現するものとして説明されている．

図 1 は概念を表す一例を示したもので，点の座標を表すモデルとその値を表示，操作できる 3 種類のオブジェクトが関連づけられている．どの GUI 部品を操作しても座標値の変化がモデルに反映すると同時に他のオブジェクトの表示を自動的に変更する．これらの部品の関連付けは支援ツールで行えるため，ソースコードに記述する必要はない．

ただし，Cocoa バインディングは Objective-C のランタイムシステムに強く依存しており，通常のオブジェクト指向言語の記述のみで実装を模倣することは容易ではない．現在の iOS 4.2 では Cocoa バインディングは利用できない．

2.3 その他のバインディング

Java を利用したソフトウェア開発では，様々なデータバインディングのフレームワークが提案されている．JavaBeans[3] はプラットフォームに依存しない Java コンポーネントを作成する仕様であり，バインディングの要素として利用されることが多い．また，Oracle ADF[4] のように，Java と XML，関係データベースをバインドし，デスクトップからモバイル環境，動的な Web ページ生成までを開発の対象とするものも現れてきた．

一方，Microsoft 社の開発環境である .Net Framework 2.0 以降では，オブジェクトのプロパティ値とコントロールのプロパティ値との間のバインディングを行うことができる[⋆1]．オブジェクトとコントロール（GUI 部品）の間の連携が主な用途である．記述言語としては，Visual Basic，C#，C++，JScript が利用できる．

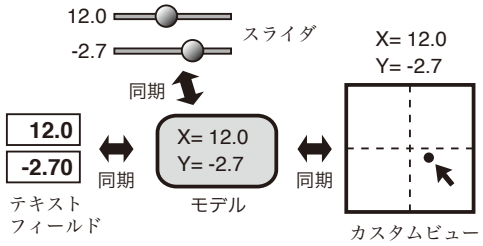


図 1 Cocoa バインディングの概念を示す例
Fig. 1 Concept of Cocoa binding.

どのバインディングのライブラリやフレームワークも，オブジェクト指向言語での利用を前提とし，さらにオブジェクトとデータソースの連携に主眼が置かれている．

3. 提案する機能の概要

3.1 coval 構造体

以下で提案する coval は，モジュール間で値を共有する仕組みを備えた一種の変数であり，実体は構造体として実装される．他の coval とバインドすることによって値が共有されるが，値を共有していても，いなくても，同じ操作で値を参照，格納できる．coval には関数を対応させておくことができ，バインド中に値が変更されると自動的に呼び出される．

例えば，図 1 の構造を表現するには，座標を値とする coval を 3 種類の GUI 部品にそれぞれ持たせておき，バインドすればよい．Cocoa バインディングと異なり，図の中央の「モデル」に相当する部分は必須ではない．同様な例を 5 節で示す．

coval 構造体の概念を**図 2**(a) に示す．構造体はバインドに必要な情報のほか，図では省略したが，起動される関数へのポインタなども含む．型 TYPE のデータを共有変数として内部に含む coval の型は，マクロによって次のように表す．

```
CovalTypeOf(TYPE)
```

⋆1 .NET Framework SDK 2.0 クラスライブラリリファレンス System.Windows.Forms.Binding クラス
http://msdn.microsoft.com/ja-jp/library/0sawzdk3(v=VS.80).aspx

Vol.2011-SE-171 No.23
2011/3/15

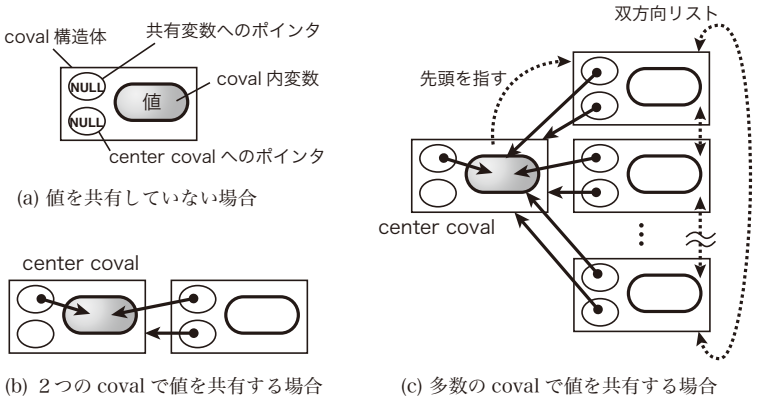


図 2 coval 構造体とバインドの概念
Fig. 2 Illustration of coval structures and binding.

変数 cov を，int 型データを共有するための coval として宣言するには次のようにする．

```
CovalTypeOf(int) cov;
```

型の別名を定義しておいてもよい．

```
typedef CovalTypeOf(int) coval_int;
coval_int cov;
```

構造体内部の実装の詳細に，プログラムが直接アクセスしないように，変数の値の参照，変更はマクロを介して行う．上の coval 型変数に含まれる int 型のデータを参照するには次のマクロを使う．このマクロは式として利用できる．

```
CovalValue(&cov)
```

値を代入するには次のようにする．この記述自体は式としては扱えない．

```
CovalAssign(&cov, i+10);
```

3.2 バインドとアンバインド

同じ型の共有変数を持つ coval 変数の間で，値を共有するためにバインドを行うことができる．2つの coval 変数 a と b があったとき，バインドは次のように行う．

```
BindCoval(&a, &b);
```

共有される値は，変数 a が保持している値になる．バインドの後，変数 a または b を使って共有変数の値を変更すると，どちらからも同じ値に変更されたように見える．

図 2(b) は2つの coval 構造体が値を共有する例である．現在の実装では，共有変数へのポインタが NULL かどうかで，バインドされているかどうかが判断できる．coval の値の参照に伴うオーバーヘッドは，このポインタを条件とした分岐だけである．

バインドされた coval の集合を coval グループと呼ぶ．coval グループ内には，共有値を保持し，動作を管理するための coval が1つ存在し，center coval と呼ばれる．変数をバインドする時，第1引数の変数はすでにバインドされていてもよいが，第2引数の変数はバインドされていてはいけない．どちらもバインドされていなければ第1引数側が center coval となる．なお，coval グループに新しい coval をバインドする場合，BindCoval の第1引数は coval グループのいずれかの coval であればよい．

バインドされた coval 変数 b を，共有関係から取り除く（アンバインド）するには次のようにする．

```
UnbindCoval(&b);
```

ただし，center coval をアンバインドすることはできない．

3.3 コールバック関数の指定

coval 変数のそれぞれについて，共有変数の値が変更された時に呼び出されるコールバック関数を指定できる．関数を指定しなければ，値が変更されても，その coval については何も起きない．

関数は次の形式のものになる．第1引数は，この関数を指定した coval 変数へのポインタ，第2引数は任意のポインタである．

```
int 関数名 (coval へのポインタ, void *)
```

関数の設定は次のように行う．引数 fn は関数ポインタ，ptr は任意のポインタで，関数が起動される時の第2引数として渡される．複数の coval に対して同じ関数を指定しても，引数によって動作を変えることができる．

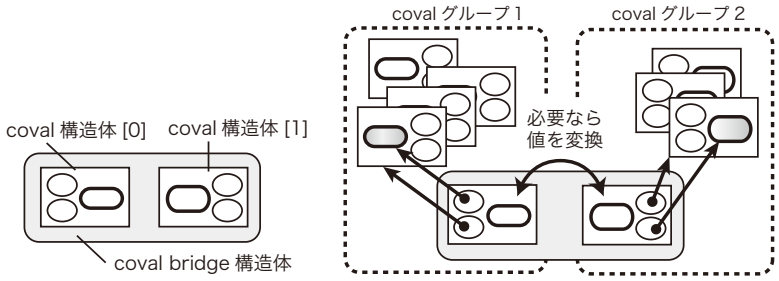
```
SetCovalCallback(&cov, fn, ptr);
```

図 2(c) は複数の coval がバインドされている状態を示している．コールバック関数が設定されている coval はリストで管理される．上で述べた CovalAssign マクロによって共有変数の値が変更された場合，coval に設定された関数がリストの順番に呼び出される．

コールバック関数は coval をバインドする前に設定しておくことができるため，実装をモジュールの内部に隠蔽することもできる．

3.4 coval bridge

coval によって共有される値は，同じ型でなければならない．しかし，同じ意味を持ち，共有したい情報が，あるモジュールでは int 型，別のモジュールでは short 型や float 型で



(a) coval bridge 構造体の概念　(b) coval bridge による共有の概念

図 3 coval bridge 構造体と coval グループの結合
Fig. 3 Binding of coval groups using a coval bridge structure.

プログラムされているという場合も珍しくない．構造体の一部のメンバだけを共有したい場合もある．

また，すでに存在する coval グループと，別の coval グループを新たに連携させたい場合，バインドの方法を変更するのではなく，グループとグループを結合できればモジュール群としての独立性を向上させることができる．

このために，coval bridge という構造を導入する．coval bridge は**図 3**(a) のように coval を 2 つ含む構造で，それぞれの coval の型は異なっていてもよい．図 3(b) のように，2 つの coval は別々の coval グループにバインドし，一方で変更された値を他方に伝えることができる．この際，必要に応じて値を変換して伝えることができる．

データ型 T1 と T2 の coval を持つ coval bridge のデータ型は次のように宣言される．

```
CovalBridgeOf(T1, T2)
```

coval bridge 型の変数 brg の持つ coval を別の coval 変数 cov とバインドするには次のようにする．なお，coval bridge 内の coval は，インデックス 0 と 1 で表現する（次の例は 0 を指定）．

```
BridgeCoval(&brg, 0, &cov);
```

2 つの coval のデータ型が異なる場合，整数型同士，あるいは整数と実数のような単純な型変換であれば coval bridge に指定して変換を行わせることができる．真偽値を否定して伝えることもできる．それ以外の複雑な場合には，変換を行うための関数を指定する．また，一方から他方へ，値の変化を通知しないように設定することもできる．

3.5 構造体を共有する場合

coval が保持する値が構造体の場合，メンバを指定して参照，変更が行える．例えば age というメンバの参照，変更は次のようになる．

```
CovalMemberValue(&cov, age) /* 式 */
CovalAssignMember(&cov, age, 19);
```

また，構造体を保持する coval に coval bridge をバインドする場合，その構造体の特定のメンバだけを指定してバインドすることができる．変数 cov が構造体を値とする coval で，coval bridge と共有したいメンバが age だった場合は次のようにする．

```
BridgeCovalMember(&brg, 0, &cov, age);
```

バインドの際に指定した以外のメンバの値が変更された場合に，coval bridge のもう一方の coval グループに変更を通知するかどうかは，coval bridge で設定することができる．

4. 実装と動作

4.1 コールバック関数の呼び出し方法

coval に対して共有変数の変更を行うと，その coval グループの中で設定されているコールバック関数が次々に呼び出される．コールバック関数は，バインドされた時期の早いものから順番に呼び出される．この一連の関数呼び出しを，以下ではコールバックループと呼ぶ．ただし，その変更を行った（CovalAssign の引数として指定された）coval に対しては，コールバック関数は起動されない（起動されるように設定することもできる）．

デフォルトでは，coval グループ内のコールバック関数はすべて順番に呼び出されるが，途中で動作を打ち切るように設定することもできる．その場合，まず center coval に中断可能であることを設定しておく．コールバック関数の返り値が，マクロ `COVAL_PROC_IGNORED` で定義された値であれば，次のコールバック関数が呼び出される．いずれかの関数が `COVAL_PROC_SUCCESS` を値として返した時，コールバックループは終了し，それ以降のコールバック関数は呼び出されない．

4.2 コールバック関数の連鎖呼び出しについて

共有変数の変更により，coval がコールバック関数を呼び出した結果として，直接，あるいは間接的にその共有変数の値が再度変更されてしまう場合がありうる．その場合にコールバック関数の再帰的な呼び出しを許すと，共有変数の変更とコールバック関数の起動が繰り返し発生し，スタックがあふれてしまう．そこで，コールバックループの間に共有変数を変更することは許すが，コールバック関数は新たに起動しない．

まず，コールバックループの開始から終了までは，処理中であることを示すフラグを center coval 内に立てておく．CovalAssign による共有変数の更新が発生した場合，このフラグを見ればコールバックループの処理中かどうか判断できる．フラグが立っていた場合，共有変数の値は書き換えるが，コールバック関数は起動しないで処理を終わる．

この方法では，連鎖呼び出しを防止することができるものの，更新した値をすべての coval に対して通知できない可能性があることに注意する必要がある．

4.3 実装とオーバーヘッド

coval の機能は C 言語のライブラリとして実装した．標準ライブラリも含めて，他のライブラリには一切依存していない．このため，標準的なライブラリが整っていない組込み機器のコードに含めて動作させることも容易である．実際に，市販されている実験用マイコン基板[5] のプログラムに含めて，動作することを確認した．

実行に必要なメモリは，コード部分以外には，各 coval および coval bridge 構造体に割り当てられる分だけである．32 ビットモデルの Intel チップ向けのコードを gcc コンパイラで生成した場合，int 型を保持する coval 構造体は 32 バイトで済む．同様に，int 型を 2 つ持つ coval bridge 構造体は 88 バイトであった．コンパイル後のコード（テキスト）は約 6 キロバイト，coval bridge を除いた場合は約 1.5 キロバイトである．

実行の際は，値を更新し，コールバック関数を呼び出すために coval center で処理を行う部分がオーバーヘッドとなりうる．

5. プログラム例

5.1 coval で GUI 部品をバインドする

図 4 に，以下で説明するサンプルプログラムの表示部分を示す．このプログラムは 2 次関数 $y = ax^2 + bx + c$ のグラフを描画するもので，式中の係数 a，b，c はウィンドウ下部のスライダ，またはテキストフィールドを用いて指定できる．また，描画領域をドラッグしてグラフを上下左右に移動させることができる．グラフの移動に伴って係数 a，b，c の値が変化するが，その変化した値はスライダとテキストフィールドの表示に反映される．

このプログラムは文献 6) の Cocoa バインディングの例題であるが，ここでは Cocoa バインディングの代わりに coval を使って部品間の連携が可能であることを示す．記述言語は Objective-C であるが，Objective-C は C 言語の上位互換の言語であるため，インスタンスオブジェクトと C の関数のやりとりに注意すれば，特に問題なく，coval の記述を含めることができる．

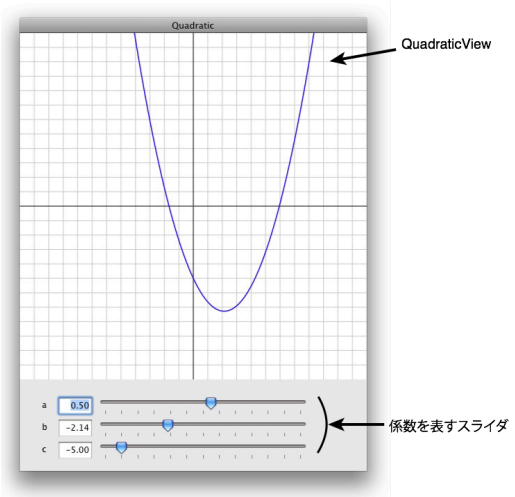
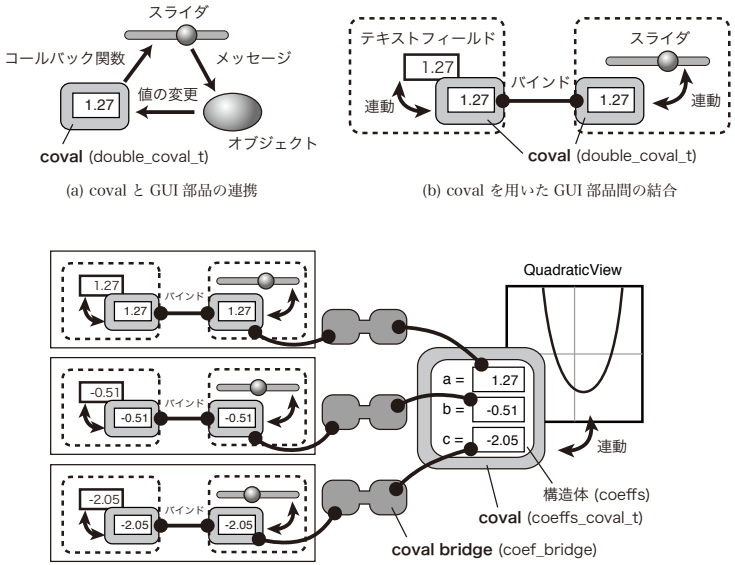


図 4 サンプルプログラムのユーザインタフェース
Fig. 4 User interface of the sample program.

図 5(a) はスライダと coval を連携させる部分の概念を示している．スライダが操作されると，ターゲット-アクションパラダイムに基づいて，指定したオブジェクトに変更を知らせるメッセージが送られる．オブジェクトはこの時，coval の値を変更する．また，coval の共有変数が変更された時には，コールバック関数の中からスライダに値を設定する．既存の GUI 部品であるスライダやテキストフィールドなどは，NSContorl というクラスのサブクラスであるため，NSContorl でこの仕組みをコーディングしておけば，その他の GUI 部品でも coval が利用できるようになる．

図 6 と**図 7** に，図 5(a) に相当するプログラムを示す．図 7 は GUI 部品と coval の仲立ちとなるオブジェクトの定義の主な部分である．このオブジェクトはインスタンス変数として myCoval という coval 変数を持つ．GUI 部品が操作されるとメソッド notifyChange: が起動されるので，ここで coval の値を変更する．coval の値が変更されるとコールバック関数である set_double() が呼び出されるので，GUI 部品に値を設定する．

なお，ここでは coval の動作の説明のために，Objective-C の既存の GUI 部品と coval を連携させる例を使っているだけであり，coval を動作させるために常にこのような補助的なオブジェクトが必要となるわけではないことに注意して頂きたい．



(a) coval と GUI 部品の連携　(b) coval を用いた GUI 部品間の結合

(c) coval bridge を用いた coval グループ間の結合

図 5 サンプルプログラムにおける coval の利用
Fig. 5 Coval structures in the sample program.

図 5(b) は，coval を介してスライダとテキストフィールドを結合させる部分の概念を示している．**図 8** の上部がこの部分のバインドに相当し，バインドするだけでスライダとテキストフィールドの表示を同期させることができる．

図 5(c) は，描画用のビュー（QuadraticView クラス）とスライダ，テキストフィールドを結合して動作させる概念図となっている．描画用ビューは 2 次関数の 3 つの係数をパラメータとして曲線を描画する．また，ビュー上でマウスをドラッグさせると，それに応じて係数値を変更する．係数値は，構造体を値とする coval に格納されており，値の変化と同期してビューの描画が行われる．

スライダとテキストフィールドをバインドした coval グループと，ビューの持つ coval のメンバを coval bridge でバインドすると，各係数値の変更をグラフの描画に反映させ，ビュー上のマウス操作によってスライダとテキストフィールドの表示を変更させることができるようになる．図 8 の後半がこの部分のバインドに相当する．

```
typedef struct _coeffs {  double a, b, c;  } coeffs;  // 3 つの係数を表す構造体
typedef CovalTypeOf(coeffs)   coeffs_coval_t;        // 構造体を値とする coval
typedef CovalTypeOf(double)   double_coval_t;        // double 型を値とする coval
typedef CovalBridgeOf(double, double) coef_bridge;  // double 型の coval bridge
```

図 6 サンプルで使われる型定義
Fig. 6 Type definition used in the sample program.

```
static int set_double(double_coval_t *cov, NSControl *cntrl)
{   // コールバック関数
    double v = CovalValue(cov);  // coval から共有変数の値を得る
    [cntrl setDoubleValue:v];    // GUI 部品に実数値を設定する
    return COVAL_PROC_SUCCESS;
}
- (void)notifyChange:(id)sender {     // GUI 部品から送られるメッセージ
    double v = [sender doubleValue];  // GUI 部品から実数値を得る
    CovalAssign(&myCoval, v);         // coval に値を設定する
}
// オブジェクトの初期化メソッド
- (id)initWithControl:(NSControl *)cntrl double:(double)val {
    if ((self = [super init]) != nil) {
        CovalAssign(&myCoval, val);  // インスタンス変数の coval に初期値を設定
        [cntrl setTarget:self];      // 自分を GUI 部品のターゲットに
        [cntrl setAction:@selector(notifyChange:)]; // メッセージの設定
        SetCovalCallback(&myCoval, set_double, cntrl);
        // coval にコールバック関数を設定．第 3 引数には GUI 部品が渡される
    }
    return self;
}
```

図 7 オブジェクトと coval の連携の例
Fig. 7 An example of combination of objects and a coval.

Vol.2011-SE-171 No.23
2011/3/15

```
for (i = 0; i < 3; i++) {
    double_coval_t *txcov = [tx[i] doubleCoval];  // テキストフィールドと
    double_coval_t *slcov = [sl[i] doubleCoval];  // スライダの coval を
    BindCoval(txcov, slcov);                      // ひとつずつバインドする
}
// qv はグラフの描画ビュー，qvcov は構造体を持つ coval である
coeffs_coval_t *qvcov = [qv covalOfCoefficients];
CovalBridgeInit(double, double, &cov_a); // coval bridge の cov_a を初期化する
BridgeCovalMember(&cov_a, 0, qvcov, a);  // qvcov のメンバ a を指定してバインド
txcov = [tx[0] doubleCoval];    // テキストフィールドの coval グループを
BridgeCoval(&cov_a, 1, txcov);  // coval bridge のもう一方とバインド
// 係数 b, c についても，coval bridge の cov_b, cov_c を使って同様にバインドする
```

図 8 coval および coval bridge の結合の例
Fig. 8 An example of binding of covals and coval bridges.

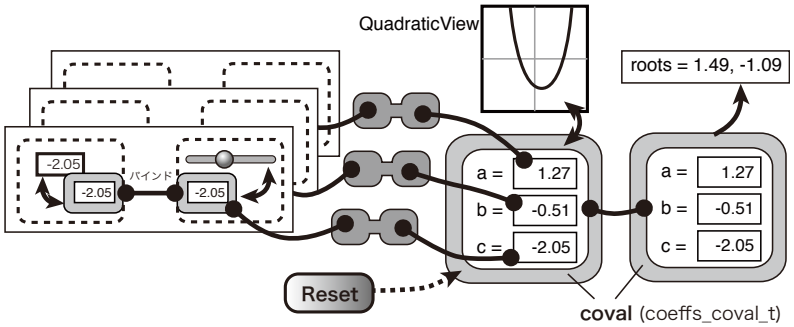


図 9 サンプルプログラムの拡張
Fig. 9 Extension of the sample program.

5.2 coval を利用して機能を拡張する

coval を使ってモジュールを組み合わせているプログラムは，機能の追加も容易である．

前節のプログラムで，2次方程式 $ax^2 + bx + c = 0$ の解を表示するようにプログラムを拡張することを考える．このためには**図 9** のように，係数を持つ coval と，その値を使って解を表示するコールバック関数を用意しておき，既存の coval とバインドするだけでよい．係数の変化に合わせて，解を次々に表示させることができる．

このプログラムを起動した時，初期値として $y = x^2 - 1$ のグラフが表示される．パラメータを変化させた後で，表示をこの状態に戻すためにリセットボタンを付けるとする．その場合，ボタンが押された時の処理として，coval に初期値を設定しなおすようにするだけでよい．グラフも，その他の GUI 部品の表示も自動的に変更できる．

6. 議　　論

6.1 モジュールの独立性向上について

本稿で提案した coval を用いるプログラミングでは，共有すべき変数をあらかじめ想定して，各モジュールに coval を利用するコードを記述しておく必要がある．基本的には，モジュールの外部からアクセス可能なプロパティを coval として記述し直すことになるため，モジュールの結合が変更されても，coval のコードを修正する可能性は低く抑えることができる．コールバック関数の実装をモジュール内に隠蔽することも容易である．

関数呼び出しのように一対一の関係に限定されず，複数のモジュール間で情報を共有できるため，サンプルプログラムの例のように，コードを変更せずにバインドだけで機能を拡張することが可能となっている．

6.2 デザインパターンへの応用

オブジェクト指向ソフトウェアの設計においては，デザインパターン[7] やソフトウェアアーキテクチャ[8] を利用する考え方が一般化してきている．しかし，これらの手法はオブジェクト指向を前提としているものがほとんどであり，手続き型言語を用いて開発を行う場合の参考にはなっていなかった．

以下では，coval を利用してモジュールを結合する手法によって，一部のデザインパターンを模倣できることを論じる．ただし，オブジェクト指向のデザインで前提としている継承や抽象化，動的結合は実現できないため，目的とする主な動作を実現できるかに注目している．なお，デザインパターンの名称は文献 7) に従う．

(1) Observer パターン
何らかのイベントが発生したことを，関係するモジュールに伝達するパターンである．一方向にだけコールバック関数を設定した coval を使って，同様な構造を実現できる．

(2) Chain of Responsibility パターン
鎖状につながれた複数のモジュールを順番に呼び出し，いずれかに処理の機会を与えるパターン．coval では，中断可能に設定したコールバックループを用いて実現できる．

(3) Mediator パターン
モジュール間の相互作用を仲介する仕組みを用意し，個々のモジュールの独立性を高めるパターン．coval だけではモジュール間の調整を行うことはできないが，動作を同期させたり，データの変換を行ったりすることが容易となるため，Mediator の機能を実現する上で役立つ．

(4) Adapter パターン，Decorator パターンなど
互換性のないモジュールを接続したり，モジュールに動的に付加機能を追加するパターンである．クラス継承を用いた本来のデザインパターンのように統一された実装は困難だが，これらのパターンで目的とする機能には，coval を用いて実現できるものも多い．coval はバインドするだけであり，関係するモジュールのコードに影響しないため，相互に接続したり，機能を追加したりすることは容易である．

6.3 組込みシステムへの適用

組込みシステムの種類や規模は極めて多岐にわたるが，4.3 節でも述べたように，coval の実装コードは比較的小さいため，リソースの少ない組込みシステムに導入する場合にも問題は少ない．

本稿でのサンプルプログラムのように，モジュール間の連携でプログラムを構築する方法は，イベント駆動方式で動作するシステムにおいて有効性が高い．組込みシステムには，デバイスなどからの入力やタイマによる状態遷移に基づいて設計されているものが多いため，オブジェクト指向を前提としない coval が活用できる可能性がある．今後，具体的な事例や有効性に関して検討を進めたい．

6.4 マルチスレッド環境での利用

本稿で示した実装では，coval はマルチスレッド環境での実行を前提とはしていない．

ある coval が呼び出したコールバック関数が別の coval の共有変数にアクセスし，別のコールバック関数を起動することがある．この関係は再帰的に生じる可能性もあり，共有変数に対するアクセスを相互排除で管理した場合には，デッドロックが発生する．従って，ひとつの coval グループはひとつのスレッドで実行するなど，プログラミングにおいて何らかの指針を設ける必要がある．

組込みシステムをはじめ，マルチスレッドが利用できない環境ではコルーチンが利用されることも多く，いくつかのライブラリが開発されている[9]．coval とコルーチンを組み合わせた場合，相互排除などの問題が発生しにくく，有効に利用できると期待される．

7. おわりに

手続き型言語で利用可能なモジュールのバインド手法として coval を提案し，その機能と実装を述べた．バインドによってモジュールを結合する方法を説明し，プログラムの記述例を示した．coval は先行研究[10),11)] におけるタップ機構の改良版と位置づけられるが，動的なバインドの導入によって実用的な記述能力を持つことができた．本稿ではまた，デザインパターンとの比較や，組込みシステムへの適用についても論じた．

今後は coval の機能，実装面での洗練化を進めるとともに，特に組込みシステムへの適用について，実用性の観点から検証を行う予定である．

謝辞 本研究は科学研究費（基盤 (C)22500038）の助成を受けたものである．

参考文献

1) NeXT Computer: *Next Development Tools*, Addison-Wesley (1991).
2) Apple Inc.: *Introduction to Cocoa Bindings Programming Topics*, http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaBindings/CocoaBindings.html (2007).
3) Sun Microsystems (Ed. Hamilton, Graham): *JavaBeans*, http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html (1997).
4) Oracle Corp. (O'Brien, S. and Shmeltzer, S.): Oracle ホワイト・ペーパー—Oracle Application Development Framework 概要, http://www.oracle.com/technetwork/jp/developer-tools/adf/adf-11-overview-130932-ja.pdf (2009).
5) 山崎尊永 他: 今すぐ使える！ H8 マイコン基板 増補版, CQ 出版社 (2010).
6) 荻原剛志: 詳解 Objective-C 2.0 改訂版, ソフトバンククリエイティブ (2010).
7) Gamma, E. et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*, AddisonWesley Professional (1994). （本位田, 吉田 訳: オブジェクト指向における再利用のためのデザインパターン 改訂版, ソフトバンクパブリッシング (1999)）.
8) Buschmann, F. et al.: Pattern-Oriented Software Architechture, A System of Petterns, Wiley (1996). （金沢他 訳: ソフトウェアアーキテクチャ—ソフトウェア開発のためのパターン体系, 近代科学社 (2000)）.
9) Engelschall, R. S.: Portable Multithreading, The Signal Stack Trick For User-Space Thread Creation, USENIX Annual Technical Conference 2000, pp.18–23 (2000).
10) 荻原剛志: 手続き型言語に適用可能なモジュールのバインド方式について, 情報処理学会研究報告, 2008-SE-112, pp.39–45 (2008).
11) 荻原剛志: C 言語への静的なバインド機構の実装, 京都産業大学論集（自然科学系列）Vol.40 (2011). （掲載予定）