ソフトウェア工学 100－4
（1994．9．8）

カスタマイズのためのアプリケーション構築モデル

上野 和彦　山本 隆広

NTT
ソフトウェア研究所

アプリケーションプログラムのユーザインタフェース(UI)のカスタマイズを行なう場合，Cなどプログラミング言語で記述されたソースコードを解析する必要がある．本論文で提案するアプリケーション構築モデルでは、ソースコードの解析を行なうことなくカスタマイズするために，アプリケーションをUI部品とAP機能の部品に別々に分けて記述し，UI部品とAP機能部品に渡される処理データの流れに着目して、UI部品とAP機能部品の接続方法をテキストで記述する．そして，カスタマイズをするときは、接続方法をユーザが変更することにより行なう．これにより，ソースコードを参照しなくても広い範囲のUIのカスタマイズが行なえるようになる．

Application Model for customizing

Kazuhiko UENO, Takahiro YAMAMOTO

NTT
Software Laboratories
3-9-11 Midori-cho, Musashinosi,Tokyo, 180 JAPAN

When we customize user interface of an application program, we must analyze the source code programmed with programming language such as C. To customize the application without analyzing the source code, We propose that the application is separated user interface parts from function parts, and we describe the connections user interface parts and function parts. When user want to customize the application, user can change some connections. The message flow describes the connection between user interface objects. And it describes some functions using in application. Therefore user can widely customize the user interface without source code.

1 はじめに

ワークステーションやパーソナルコンピュータの高機能化, 低価格化により多くのユーザがコンピュータを利用するようになった. そのため, 専門家でなくても利用できるようにグラフィカルユーザインタフェース (GUI) を採用するなどユーザインタフェース (UI) の改良が行われてきた. そして,Athena ウィジェット[1] や OSF/Motif ウィジェット[2] など, 多くの人にとって使いやすい UI が提案・実現されてきた. しかし, アプリケーションプログラムを利用する場合, ユーザの環境や作業内容などにより, 最適な UI は変わってくる. そのため, アプリケーションプログラムのカスタマイズが必要になる.

アプリケーションプログラムをユーザが変更しようとするとき, 通常行えるのは, アプリケーション作成時に用意された項目だけである. しかし, あらかじめ用意されていない変更を行おうとすると, プログラミング言語で書かれたソースプログラムを調べて, それを変更しなければならない. しかし, 変更したいアプリケーションのソースプログラムが必ずしも提供されているとは限らない.

そこで, 本稿では, プログラミング言語で書かれたソースプログラムを変更すること無く, カスタマイズすることのできるアプリケーションの構成方法を提案する.

ここでは, 通常のアプリケーションの処理が,

1. ユーザからの操作指令
2. データ処理
3. ユーザへの情報の出力

となっていることに着目する. 入出力のための UI 部分とデータ処理のための機能部分とにアプリケーションを分ける. そして,UI の構成部品と機能部分を独立した部品として作り, UI 部品と機能部品の接続方法を部品群と分けて記述する. そして, 接続部分の変更で UI と機能部分のつながりを変え, カスタマイズを行う.

本論文で提案した構成方法では,UI を集中的に記述でき, UI のカスタマイズをする時に機能部分のソースコードを必要としないという利点がある.

2 アプリケーションプログラムのカスタマイズ

2.1 カスタマイズ作業者

アプリケーションのカスタマイズには 2 つの場合がある. 開発者によるカスタマイズとユーザによるカスタマイズである.

開発者によるカスタマイズ

開発者によるカスタマイズの場合, カスタマイズを行う者はアプリケーションの詳細を知っている, または, 知ることができる状態でカスタマイズする. また, 通常, その作業をするのは技術力をもった人である.

ユーザによるカスタマイズ

もう一つは, ユーザがカスタマイズする場合である. この場合, 通常はユーザが自分の望む状態にアプリケーションを変更することを目的とする. 開発者が行う場合と異なり, アプリケーションの詳細を知らない状態でカスタマイズすることも多い.

例えば, あらかじめ用意してあるメニュー方式が 2 種類あり, ユーザがどちらかを選択するといった場合がある. 開発者があらかじめ用意してあるカスタマイズは, アプリケーションの詳細を知らなくても, カスタマイズ項目の選択をしたり, 環境変数やリソースファイルなどの設定をしたりすることで簡単にカスタマイズできる.

それ以外の開発側が用意していないカスタマイズの場合は, アプリケーションの詳細を知る必要がある. また, そのカスタマイズを行うユーザは, 必ずしも技術力がある人とは限らない.

2.2 ユーザによるカスタマイズの詳細

ユーザによるカスタマイズには次のようなものがある.

1. GUI の Look & Feel の変更
2. 機能の付加や組み合わせ
3. 入出力方法の変更

1 は, キーやマウスボタンのバインド, ウィジェットの色や配置, メニュー構成といったものの変更がある. これらは, X-Window システムではリソースの変更や,UIL の利用である程度のカスタマイズは可能となる.

2 は, 新しい機能の付加や, 既存機能を組み合わせて新しい機能として登録するといったカスタマイズである. これは, アドインソフトとして別パッケージで提供されていたり, マクロ言語が提供されているアプリケーションでマクロプログラムを組み, 用意された機能を組み合わせることができたりする.

3 は, マウスによるコマンド選択を音声による入力にしたり, 画面情報の音声出力といったカスタマイズである.

本稿では, 主に, 2,3 のカスタマイズを対象としたアプリケーションの構成方法を述べる.

2.3 各種のカスタマイズ方法

ここでは, 既存のカスタマイズ方法の特徴を述べる.

2.3.1 リソース

X-Window システムで GUI を使ったアプリケーションは, ほとんどの場合, ツールキット (ウィジェット) を利

用している.主なものに,Athena ウィジェットセットやOSF/Motif ウィジェットセットがある. これらツールキットのウィジェットはリソースと呼ばれる属性を持っている.各ウィジェットはリソースの値を参照することにより,見た目や振舞い (Look & Feel) を決定する.

アプリケーションプログラムの実行ファイルとは別に通常のテキストで記述されるリソースファイルを用意することでリソース値を指定できる.リソースファイルを利用すれば,ウィジェットの配置や色やキーバインドなどを実行ファイルをさわることなく,ある程度の変更ができる.

しかし,リソースではウィジェットの色や位置関係を変更したり,マウスボタンやキー入力に応じて呼び出される関数を変更することしかできない.新たにウィジェットを追加したり,メニューを作り直すといったことや,新しく機能を付け加えるといったことはできない.

2.3.2 UIL

OSF/Motif には UIL(User Interface Language) という仕様言語がある. UIL は,ウィジェットやデータの定義をテキストファイルで記述する. UIL ファイルは,バイナリ形式にコンパイルされ,アプリケーション実行時にMotif リソースマネージャルーチンにより参照され,ウィジェットが生成される.

UIL では,アプリケーションで利用するウィジェットの指定や,リソース値の指定ができる.また,ウィジェットの定義を書き加えれば, UI に新しくウィジェットを付け加えることができる.また,メニュー構成を変えるようなことも可能である.そのため,リソースによる UI のカスタマイズよりも広い範囲でカスタマイズできる.

また,UIL を使用することで, UI とアプリケーションコードの分離ができる.そのため,アプリケーションプログラムを再コンパイル,リンクしなくても,テキストファイルの変更と UIL の再コンパイルをすれば, UI を変更できる.別々の UIL ファイルを使用して,特定のロケールに合わせた UI を用意し,切替えるということもできる[3].

また,UIL に対応した UI 構築ツールもあり, UIL ファイルの作成を視覚的に行うことができる.

2.3.3 アドインソフト

アプリケーションを機能追加が可能になるように設計し,付加的な機能を後から追加できるようにしたアプリケーションがある.ユーザは提供された機能の中で必要な機能を選択して,アプリケーションに追加することができる.

2.3.4 マクロ

アプリケーションに搭載されたマクロ言語により,アプリケーションの基本機能を組み合わせて利用することが可能になる.また,別のアプリケーションを制御できるマクロ言語を搭載しているアプリケーションもある.

3 カスタマイズのためのアプリケーションモデル

ここでは本論文で提案するアプリケーション構成モデルとカスタマイズ方法について述べる.

3.1節ではアプリケーションの構成方法を述べる. 3.2節, 3.3節, 3.4節ではアプリケーションの構成要素の詳細について述べる. 3.5節ではカスタマイズ方法について述べる.

3.1 アプリケーションの構成

アプリケーションはマウスボタンクリックのようなアプリケーション外部からのイベントがあったときに処理を始め,それを外部に出力することでひとつの処理を終えると考える.すなわち,動作を開始する合図を外部から受けとる部分,処理部分,処理結果を外部に伝える部分とわけて考える.

ここでは,アプリケーションを,「ユーザインタフェース」,アプリケーションの機能部分である「アプリケーションファンクション」,それらを接続するための「メッセージフロー」で構成されているものと定義する (図1).

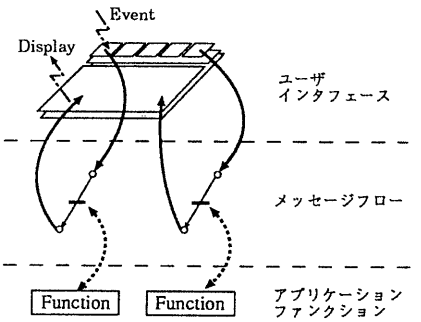

図 1: アプリケーション構成

アプリケーションは,入力用の UI 部品が外部からイベントを受けとり,それに応じて処理を始める.そして,処理結果を出力用の UI 部品で出力するという流れができる.この流れを「メッセージフロー」として定義する.

アプリケーションは,アプリケーションファンクション記述部分と UI 記述部分とメッセージフロー記述部分を分けて記述する.

3.2 ユーザインタフェースオブジェクト

UIを構成する部品をここでは,ユーザインタフェースオブジェクト(UIO)と呼ぶ.UIOの主なものは,プッシュボタンやスクロールバーなど,ウィジェットやコントロールと呼ばれるものである.また,音声の入出力を行うオブジェクトのような画面に現れないものなども含める.これらは,マウスボタンのクリックなど外部からのイベントを受けとり,フローにメッセージを送り出す.また,フローから受けとったメッセージに従い,決められた動作を行う.例えば,"show" というメッセージを受けとったら画面に現れたり,"set" というメッセージを受けとったら,メッセージを適切なリソースに設定したりするなど,いくつかの種類のメッセージを解釈し,それに応じて動作する.

UIOは,それ自身でイベントに対するアクションも持つ.例えば,プッシュボタン上でマウスボタンが押された時には,ボタンが引っ込んだり,テキストウィジェット上で,通常文字をタイプすれば,それをエコーバックしたりする.

3.3 メッセージフロー

メッセージフローは,外部からの入力をうけとるUIOと,外部への出力を行うUIOとの間のメッセージの流れる道を記述する.入力用のUIOよりメッセージを受けとり,それを出力用のUIOへ向かって流していく.また,その過程において,適切な出力が得られるようにメッセージを加工していく.

メッセージフローは,フローを区切るゲートウェイ,メッセージが流れている途中でメッセージを変換するオペレーション,メッセージを流すための条件であるゲートよりなる(図2).

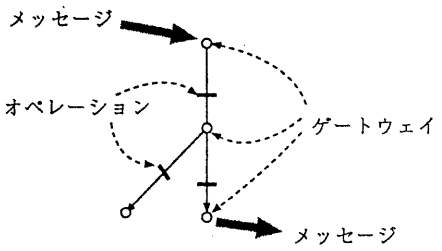

図2: メッセージフロー

メッセージフローは自由に分岐,結合できる.

結合点は,複数のフローのメッセージをつなげるのではなく,それぞれのフローからのメッセージをそれぞれ別々に流す.図3(a)のような結合があるフローと,(b)のように2つに分けたフローは等価である.AからDへ流れるメッセージはBからDへのフローからの影響は受けず,同様に,BからDへ流れるメッセージはAからDへのフローからは何の影響も受けない.

分岐する場合,通常すべてのフローにメッセージを流す.図4(a)と図4(b)のフローは,通常は等価である.ただし,共通部分のオペレーション(図中OP1)に副作用があり,処理をするたびに得られる出力が異なるような場合には,等価なものとはならない.

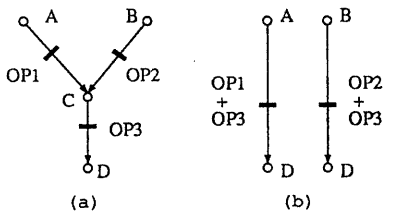

図3: フローの結合

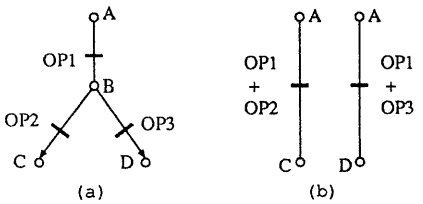

図4: フローの分岐

ゲートウェイ

ゲートウェイは,フローとUIOとの接続点,および,フローの分岐点,結合点となる.ゲートウェイ自体はフローの区切りとなるだけで,なにも機能や状態を持たず,その先のフローやUIOへメッセージを流すだけである.

ゲート

ゲートは,ゲートウェイからその先のフローへメッセージを流すための条件である.条件が成立しない場合は,そこでメッセージの流れは止められる.

オペレーション

メッセージフローには,いくつかのオペレーションが附属する.オペレーションは,流れているメッセージを適当なメッセージに変換する.メッセージの変換を行うために,アプリケーションファンクションや別アプリケーションプログラムを呼び出すこともできる.また,それらの変わりに,UIOとメッセージをやりとりすることもある.

3.4 アプリケーションファンクション

アプリケーションファンクションは,アプリケーションの機能部分である.これは,通常のプログラミング言語で記述された関数である.アプリケーションファンクションは,メッセージフローよりメッセージを受けとり,適当なメッセージに変換し,メッセージフローに戻すだけの変換器であり,UIO を直接には操作しない.また,各アプリケーションファンクションは,それぞれ完全に独立し,メッセージフローを介してのみ接続される.そのため,それぞれのアプリケーションファンクションをまるごと交換するということが容易になっている.

3.5 フローによるカスタマイズ

カスタマイズはメッセージフローと UI 記述の変更により行う.例えば,メッセージの音声出力を行う場合には,音声出力用の UIO を UI 記述に追加し,その UIO へメッセージを流すフローを追加する.

また,UIO をツールキットごと変えるとき,メッセージによる動作が元の UIO と異なる UIO でも,メッセージをオペレーションで変換することで,対応できる.

機能を追加する場合は,新たにアプリケーションファンクションを追加し,そのアプリケーションファンクションを利用するメッセージフローを追加する.外部プログラムを利用する場合はフローファイルの変更だけになる.

4 実装

4.1 動作環境

SPARC Station10, SunOS 4.1.3, X-window 上で本論文で提案したアプリケーション構築モデルを実装し,テストした.

実装を簡単にするために,UI 記述言語に UIL を使用する.そのため UIO に Motif ウィジェットを利用する.メッセージフローは独自に定めた記述仕様で記述する.アプリケーションファンクションや初期化部分などは C 言語で記述する.実行時には,メッセージフローファイルと,UIL ファイルをコンパイルして得られる UID ファイルが参照される.また,外部プログラムを実行時に利用することもある (図 5).

4.2 UI 記述

UI の記述には Motif の UIL を利用する.サンプルプログラム (図 6) の UIL ファイルは図 7のようになる.このプログラムはボタンを一度押すと,ボタンのラベルが Good-By! にかわり,もう一度押すとプログラムを終了する.また,ボタンを押すたびに,date コマンドを標準出力へ出力する.

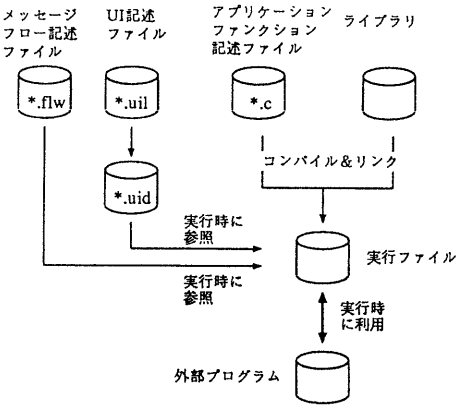

図 5: ファイル構成

図 6: サンプルプログラム

通常の UIL によるウィジェットの定義に加えて,

```
value
  fetch_widget : exported "widget, widget, ...";
  map_widget : exported "widget, widget, ...";
```

のように,アプリケーションがフェッチする (UID ファイルから読み込む) ウィジェットとウィンドウマネージャがマネージする (画面に表示する) ウィジェットを指定する.

フローファイルで利用するウィジェットは, Mrmcreate callback に create_proc を指定し,その他必要な callback は,送り出すメッセージを引数にして msm_callback を呼び出すようにする.

4.3 メッセージ

メッセージには文字列の配列を使用している.オペレーション中でのメッセージを参照するには,一番目の文字列を $1, 二番目を $2... というように記述する. $0 はメッセージ全体を意味する.

```
! Hello world

module helloworld
  version = 'v1.0'
  names = case_sensitive

value
  fetch_widget : exported "helloworld_main";
  map_widget   : exported "helloworld_main";

procedure
  msm_callback;
  create_proc();

object
  helloworld_main : XmRowColumn {
    controls {
      XmLabel       helloworld_label;
      XmPushButton  helloworld_button;
    };
  };

object
  helloworld_label : XmLabel {
    arguments {
      XmNlabelString =
        compound_string("Press button once",
                        separate=true)&
        compound_string("to change label;",
                        separate=true)&
        compound_string("twice to exit.");
    };
  };

object
  helloworld_button : XmPushButton {
    arguments {
      XmNlabelString =
        compound_string("Hello World!");
    };
    callbacks {
      XmNactivateCallback =
        procedure msm_callback("Helloworld");
      MrmNcreateCallback =
        procedure create_proc();
    };
  };
end module;
```

図 7: サンプルプログラムの UIL ファイル

4.4 メッセージフロー記述

サンプルプログラムのフローファイルの記述例は図 8のようになる. なお,"!" で始まる行はコメント行である. また, これを図示すると図 9になる.

メッセージフローの記述は,UIO 定義部とフロー定義部に分かれる.

4.4.1 UI定義部

UIO 定義部は, メッセージフローで使用する UIO 名を記述する. この記述は以下のようになる. ただし,* は 0 回以上の繰り返し, | は選択, [ ] は省略可能を意味している.

UIOdefine := UIO ['{' gateway [',' gateway]* '}'] ';'

このとき,UIO がメッセージを送る場合には 送り先のゲートウェイを列挙し指定する. UIO はメッセージを送る時には, そのゲートウェイすべてにメッセージを送る.

```
! hello.flw

!UIO 定義
object {
  helloworld_button {A_helloworld, A_date};
}

! フロー定義
flows {
  A_helloworld {$1 /Helloworld/
     {call helloworld_activate;} } A_callback;

  A_callback {$1 /label/ {"set","Good-bye!";} }
                              helloworld_button;

  A_callback {$1 /quit/ {} } MsmQuit;

  A_date { {exec "date";} } stdout;
}
```

図 8: メッセージフローファイルの記述例

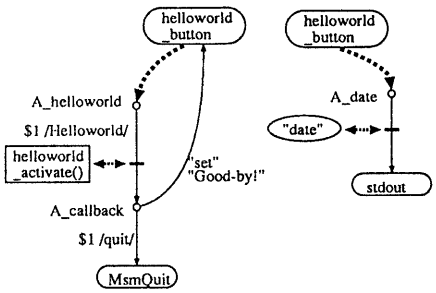

図 9: メッセージフロー例

4.4.2 フロー定義部

フロー定義部は, メッセージフローを定義する. 記述方法は以下のようになる.

flow := gateway '{' *script** '}' gateway ';'
　　| gateway '{' *script** '}' UIOname ';'
script := *gate* '{' *new-message** '}'
gate := [$n] ['/' pattern '/']
new-message := *operation* [','*operation*]*';'
operation := 'call' function message* ';'
　　| 'call' UIO string*
　　| 'exec' "shell command" string*';'
　　| string';'

gate は, メッセージ中に pattern が含まれるかどうかをチェックする. ここで,$n は, メッセージの n 番目の文字列と pattern とを比較する. $n が省略された時にはメッセージのすべての文字列との比較を行う.

operation は,"call" 命令, "exec" 命令, または, 文字列が書かれる. "call" 命令はアプリケーションファンクションを呼び出すか, 戻ってくるメッセージが存在する UIO へメッセージを送ることができる. "call" に続けて関数名または UIO 名をかき, その後に送るメッセージを記述する. "exec" 命令は外部プログラムを起動する. "exec"

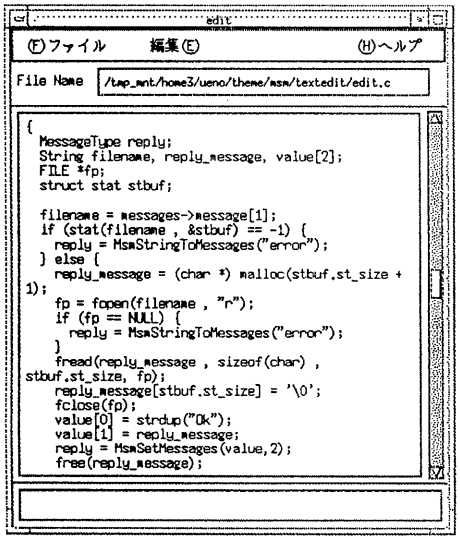

図 10: テキストエディタ

に続けて, プログラム名を引数を含めて記述し, それに続けて, 外部プログラムの標準入力に送るメッセージを記述する. また, 外部プログラムの標準出力を新しいメッセージとして受けとる.

message はダブルクオートで囲まれた文字列である. この文字列のなかに "$n" を含めると, "$n" の部分を, メッセージの n 番目の文字列に置き換える.

メッセージを UIO に送る時には, 送り先ゲートウェイの変わりに UIO 名を記述する. UIL で定義しない UIO として, "MsmQuit", "stdout" がある. MsmQuit にメッセージを送ると, 他のすべてのメッセージを送り終えた後で, プログラムを終了する. stdout は標準出力へメッセージを出力する UIO である.

4.5 実験例題

サンプルプログラムとして, テキストエディタを作成した (図 10). また, 同時に, 同じものを C 言語と UIL のみを用いて作成し, 比較した.

プログラムサイズは, 以下のようになった.

- フローを利用したプログラム
  - メッセージフローファイル:104 行, 約 4Kbyte
  - UIL ファイル:639 行, 約 17Kbyte
  - アプリケーション機能部分 (初期化ルーチンを含む):C 言語:241 行, 約 6Kbyte
- フローを利用しないプログラム
  - UIL ファイル:650 行, 約 18Kbyte
  - アプリケーション機能部分 (初期化ルーチンを含む):C 言語:398 行, 約 10Kbyte

4.6 カスタマイズ実験

今回の実験では, 音声によるオペレーションを想定した実験を行った. ただし,Motif ウィジェットに音声を出力するものが無いため, その替わりとして, 別のウィンドウ上のテキストウィジェット v_object を出力 UIO に使用した.

カスタマイズ項目は以下の通りである.

- メニュー項目選択時に選択カーソルがある項目の読み上げ
- 通常, メッセージ出力領域に出力されるメッセージの読み上げ
- ファイル名の出力用ウィンドウにフォーカスが来た時に行う, ファイル名の読み上げ

例えば, メニュー項目の読み上げでは,UIL ファイルを変更し, メニュー項目に選択カーソルが来たときにコールバックが起こるようにする. そして, 図 11のように, メッセージを流すフローを作る. この例では search のボタンの例を示している. 図左側がサーチ用ののダイアログを開くためのもともとのフロー. 右側が新しく追加したフローである.

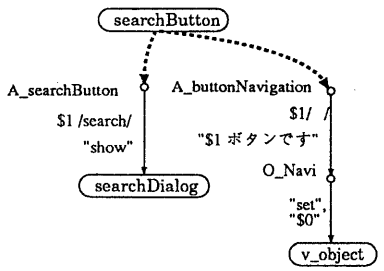

図 11: ナビゲーションのフロー

4.7 カスタマイズ実験結果

このカスタマイズで変更行数は表 1のようになる. フローファイルの変更は,

- UIO 定義部:11 行
  音声出力のための新しい UIO の定義と, メニュー項目などのからフローファイルに音声用のメッセージを送るようにする変更.

- フロー定義部:6 行
  新しい UIO へメッセージを流すメッセージフローの追加.

UIL ファイルの変更は,

- フェッチとマネージのための新しい UIOの追加:2 行
  音声出力用 UIO(v_object) のフェッチとマネージ.
- コールバックの指定:11 行
  メニュー項目などの UIO からフローファイルにメッセージを送るようにするためのコールバック.
- 新しい UIO(v_object など) の定義:22 行

となった.C プログラムの変更はない.

C と UIL のみを使用した場合の変更は, UIL ファイルの変更は,

- コールバック関数の登録:2 行
- 新しい UIO(v_object) の ID 番号指定の追加:1 行
- コールバックの指定:11 行
- 新しい UIO(v_object など) の定義:22 行

C プログラムは,

- 音声出力のためのコールバック関数の記述:48 行

となった.

表 1: 変更行数

	C + UIL + FLW (変更行 / 全行数)	C + UIL (変更行 / 全行数)
フローファイル	17/111	—
UIL ファイル	35/673	36/686
C プログラム	0/241	48/446

5 評価

メッセージフローには以下の特徴がある.

- UI の部分を集中して記述できる. これにより,UI のみの変更の場合, C プログラムの変更, コンパイルが不要になる.
- 外部プログラムを利用する時も, フローファイルの変更のみでよく, C プログラムを変更する必要がない.
- アプリケーションの機能部分をフィルターとして作ることで, 機能部分に UI の記述がなくなり UI の変更に対する影響がなくなる.

また, 以下のような問題点があげられる.

- 動的に UIO, フローを作ることができない.
- 処理の途中で GUI にアクセスすることができない. 例えば, 長い処理の途中経過を示すといったことが基本的にできない.
- アプリケーションファンクションを呼び出す時のオーバーヘッドが大きい.

6 まとめ

本稿では,GUI の記述とアプリケーションの機能部分の記述を分け, そのつながりをフローファイルで記述する方法を示した. アプリケーション機能部分をフィルターとして作り, その部分では,UI にアクセスする関数を記述しないようにすることにより, アプリケーションの機能に関係なく UI の変更ができるようになった.

今後の課題として以下のことを検討している.

- 視覚的にメッセージフローを記述するツールの作成. GUI の構築と, メッセージフローの作成を視覚的に行うツールを提供することで, カスタマイズの手間が減り, また, 流れが理解しやすくなる.
- 動的な UIO の導入.
  アプリケーションファンクションを変更しないままで, UI 記述言語で定義された UIO をテンプレートとする UIO を動的に作り, 利用できるようにする.

謝 辞

本研究において, 助言していただいたＮＴＴソフトウェア株式会社の黒木宏明氏に深く感謝します.

参 考 文 献

1) 安居院猛, 永江孝規 “X アプリケーション・プログラミング 2 Athena ウィジェット編”, 新紀元社, 1992
2) 兜木昭男, 木下凌一, 栄谷政己, 林秀幸, 安川悦子, “X-Window OSF/Motif プログラミング”, 日刊工業新聞社,1990
3) Open Software Foundation “OSF/Motif プログラマーズガイド リリース 1.2”, 株式会社トッパン, 1993
4) 守谷慎次 “ユーザインタフェース管理システムの基本概念および対話の方式”, 情報処理 Vol.33 No.11, pp. 1285–1294, 1992
5) 上野和彦, 山本隆広, 黒木宏明, “カスタマイジングを容易にする UI の構築方法”, 情報処理学会第 48 回全国大会, pp.5-231–5-232, 1994