ソフトウェア工学　74－3
（１９９０．　９．　１１）

SUIT：ソフトウェア・ユーザーインターフェイス生成ツール

来住伸子　佐藤和夫　鈴木美幸　伊藤圭一　平野一路
日本アイ・ビー・エム株式会社

SUITは，仕様レビュー段階で使用することを主な目的とするグラフィカルユーザーインターフェイスの設計作成ツールである．操作構造の視覚化により，直接操作によるユーザーインターフェイス作成の簡素化と，ユーザー主導型のユーザーインターフェイス設計の支援を実現した．操作構造の視覚化は，イベント駆動型プログラムの振る舞いの解析により行い，各ウィンドウをノード，ウィンドウ間の制御の移動の可能性をフローとして表現したものである．

SUIT: Software User Interface generation Tool

*Nobuko Kishi, +Kazuo Satoh, +Yoshiyuki Suzuki, +Keiichi Itoh, +Kazumichi Hirano
*IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku, Tokyo 102, Japan
+Product Assurance Laboratory, IBM Japan, Ltd.
1623-14, Shimatsuruma, Yamato-shi, Kanagawa-ken 242, Japan

SUIT, a Software User Interface generation Tool, is developed for prototyping graphical user interfaces at early stages of software development. It offers a visualization of window relationship in a given user interface design to enable easier modification of the design and review of design techniques. The visualization is implemented by analyzing an event-driven program and by showing windows as nodes, and possible transfer of controls among them as flows.

0. はじめに

SUITは，ソフトウェア開発の初期段階，仕様レビュー段階で，グラフィカルユーザーインターフェイスのプロトタイプの作成および評価を容易にするためのユーザーインターフェイス作成ツールである．コーディングのような後期段階ではなく，初期にプロトタイピングと評価を行う必要性については，いくつかの研究があるので[1]，ここでは，SUIT開発の背景と設計目標，実現環境と方法，簡単な使用例を紹介する．

1. 背景

1.1. グラフィカルユーザーインターフェイスにおける操作構造

グラフィカルユーザーインターフェイスの長所として，グラフィカルな表現の使用によるわかり易さの向上と，ユーザー主導型の操作構造の提供の２点が挙げられる．グラフィカルユーザーインターフェイスの設計指針では，これらの長所を利用して，ユーザーに必要な機能を，分かりやすく簡単な操作として提供することを勧めている[2]．

これらの長所のうち，グラフィカルな表現については利用が進んできたが，ユーザー主導型の操作構造については，必ずしも十分に提供されていないのが現状である．

ユーザー主導型の操作構造とは，ユーザーと計算機の対話の進行状況をユーザーが制御できるような対話処理を意味する．ユーザーのどんな入力に対しても，システムは何らかの応答をし，かつその応答がユーザーの期待に沿うことが必要になってくる．そのための技法は様々だが，単純なものとしては，

1)モードを少なくする．

2)ユーザーが同時に使用できる操作の種類を増やす．

3)取消やヘルプのような標準的な機能はいつでも使用できるようにする．

などが挙げられる．

しかし，これらの単純な技法は，使用すれば必ず使い易いユーザーインターフェイスになるというものではない．たとえば，モードを少なくするという技法とは逆に，異なるモードを利用することによって，修正の難しいユーザーの誤操作を未然に防ぐ，操作の効率を上げるなどが可能になることもある．

従って，ユーザー主導型の操作構造の提供には，上述のような技法の使用を助けるツールや，プロトタイプを使用して，実際にそれらの効果を確かめる手段が必要だといえる．

1.2. イベント駆動型プログラミング

グラフィカルユーザーインターフェイスのプロトタイプ作成には，ウィンドウシステムについての知識と，プログラミンの手間がかなり必要である．ウィンドウ画面エディタのようなツールは，グラフィカル表現の使用の簡単化という面では役立っているが，操作構造の設計作成については，イベント駆動型のプログラムをＣ言語などで書くという方法しか提供されていないのが現状である．

イベント駆動型プログラミングは，ユーザー入力と計算処理の同期をとる手法として，グラフィックスアプリケーションなどで使用され始めた[3]．イベント駆動型プログラムの厳密な定義は，特にないが，次のような構成要素をもつプログラムを意味することが多い．

a. イベント生成部(event generator)
キーボードやマウスといった入力デバイスドライバやプロセスに対応するもので，複数同時に存在しても構わない．

b. イベント列(event queue)
イベント生成部が生成したイベントを保持しておくためのバッファ領域．

c. イベント振り分け部(event dispatcher)
イベント列から，イベントを取り出し，各々のイベントを適当な処理プロセス，イベント処理部(event handler)に渡す．

このような構成要素を，プロセス間通信の機能を提供しているオペレーティングシステム上に作成することは可能であり，実際，多くのウィンドウシステムのAPIは，イベント駆動型のプログラミ

ングを前提として提供されてている．

しかし，これらのAPIでは，ユーザーインターフェイス作成の容易さという面でまだまだ改善すべき点が多い．たとえば，プロセスの生成やイベント列の生成というような作業をユーザーインターフェイス以外の部分と混在して行うことになるので，学習の容易さ，プログラムのわかり易さ，保守性等が問題になり易い．そのため，ユーザーインターフェイスの操作構造を設計および変更するときに，ウィンドウ間の移動の操作の提供を怠ってしまったり，操作全体からみた一貫性を保持することを忘れてしまったりすることがある．

2. SUITの設計目標

以上のような背景を踏まえ，グラフィカルユーザーインターフェイス作成のために，SUITでは，操作構造の直接操作とユーザー主導型ユーザーインターフェイスの設計支援の２点に設計目標をおいた．そして，それらを実現するために，操作構造の視覚化をイベント駆動型プログラムの振る舞いを分析することにより行った．

2.1. 操作構造の直接操作

ユーザーインターフェイス作成を容易にするために，ウィンドウ間を移動する標準的な操作は，後述するノードとフローによって表現される操作構造を直接操作することによって変更できるようにした．たとえば，あるウィンドウを開く操作を押しボタンからプルダウンメニュウに変更したり，ある操作で開くウィンドウの種類を変更するというようなことを，ノードに対応するアイコンと，フローに対応する矢印を操作によって行える．複雑な変更は，テキスト表現を利用して行うので，視覚プログラミング環境とは異なる．一定の設計指針にもとづいたユーザーインターフェイスでは，ウィンドウ間を移動する操作の種類には指定があるので，標準的な操作を直接操作できるだけでも，かなりの操作構造の変更が行える．

2.2. ユーザー主導型ユーザーインターフェイスの設計支援

操作構造の視覚化によって，ユーザーに使用可能なウィンドウの一覧およびそれらの間の関係が図示される．そこで，ユーザー主導型の操作構造にするための技法のうち，以下のものの使用状況を簡単に調べられるようになった．

(1)ユーザーがある作業を行うのに使用しなくてはいけないウィンドウは何個あるか．

(2)各ウィンドウはどんな種類でどんな機能を提供しているか．

(2)各ウィンドウ間の移動のためにどんな操作が提供されているか．

(3)各ウィンドウに，ヘルプ・取消など標準的に必要とされる機能が提供されているかどうか．

操作構造の視覚化では，評価しにくい項目，例えば，各ウィンドウでの表示が分かりやすいか，使い易い操作かどうかというような項目に付いては，SUITで作成されるプロトタイプを実際に使用してみることによって，評価することができる．

3. SUITの実現環境

SUITが対象とするアプリーケーションはOS/2 プレゼンテーションマネジャのアプリケーションである．そこで，SUIT自身も，OS/2 プレゼンテーションマネジャのアプリケーションとして開発した．そして，EASEL社のEASEL for OS/2 EEという開発システムが，画面設計ツールを提供しており，イベント駆動型のユーザーインターフェースプログラムもEASEL言語によって記述できるので，利用することにした[4]．EASELは，GUIだけでなく，通信機能などもサポートするアプリケーション開発システムで，EASEL言語で記述されたプログラムの実行環境と，EASEL言語のプログラムを出力する画面設計エディタなどの開発環境とで構成される．

SUITは，操作構造エディタを中心にしたユーザーインターフェイスを提供し，SUITの操作構造エディタの中から，他の構成要素，EASELの画面設計エディタ，EASELコンパイラ，EASELの実行環境等をユーザーが起動できるようになっている（図１）．

SUITの操作構造エディタは，EASELの画面設計

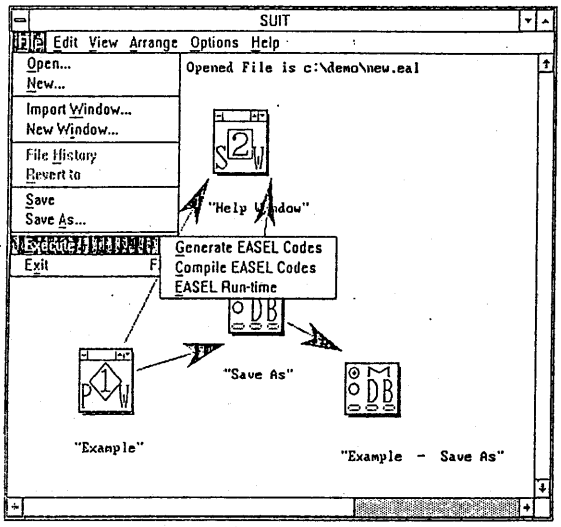

図１．SUITの操作構造エディタの画面例

エディタの出力したEASELプログラムを読み込み，操作構造の作成変更をした後，再びEASELプログラムを出力する．出力されたEASELプログラムは，コンパイルするとユーザーインターフェースのプロトタイプとして使用できる．

4. SUITにおける操作構造の視覚化のしくみ

4.1. EASEL言語によるイベント駆動型プログラミング

EASELで使用されているEASEL言語は，プレゼンテーションマネジャが提供する機能のうち，標準的なユーザーインターフェイス作成に必要な機能を言語組み込みの機能として提供している．

オブジェクトの生成については，ウィンドウを，一次ウィンドウ(primary region)，二次ウィンドウ(region)，ダイアログボックス(dialog box)という各タイプのウィンドウに対応する予約語によって，オブジェクトとして生成できる．ユーザーインターフェィス要素では，押しボタン(push button)，アクションバー(action bar)，入力フィールド(entry field)などを，やはり予約語によって容易に生成できる．

さらに，イベントについては，レスポンス文(response to ...)によって，画面上のオブジェクトをマウスの選択や，キーボードからの入力によって起きるイベント処理を簡単に定義出来るようになっている．

Listing 1 は，EASELプログラムの一例の抜粋である．プログラムの前半部にオブジェクトの宣言部があり，続いて，手続きの宣言やレスポンス文の宣言がある．

EASELはイベント処理を二つの手段で制御する．一つは，オブジェクトの属性によって制御する方法で，ダイアログボックスをモード付きにするかしないかに対応する．

もう一つの方法は，ガード付きのブロック構造を利用する．EASELのブロック構造は，レスポンス文の有効期間を制御するためのものである．一つのブロックは，予約語，beginとendで囲まれており，複数のレスポンス文を含むことができる．あるイベントが起きたとき，アクティブなブロックに含まれる，そのイベントに対応するレスポンスが実行される．ブロックが重なっているとき，ガードの付きのブロック構造の場合は，最も内側のブロックのみがアクティブになり，ガードが付いていない場合は，外側のブロック構造もアクティブの状態になっている．

4.2. ノードとフローによるイベント駆動型プログラムの視覚化

SUITでは，操作手順の全体像を視覚化するために，EASEL言語で記述されたプログラムから，ウィンドウの種類とウィンドウを開く処理，ウィンドウを閉じる処理にあたる情報を抽出し，ノードとフローで表現することにした．

一次・二次ウインドウ，ダイアログボックスといった，ウィンドウに対応するオブジェクトをノードとした．これらのオブジェクトは画面を書き換える面積が大きいので，ユーザーインターフェースに与える影響が大きいと考えられるためである．

各ノードの種類は，アイコンによって区別できるようにした．図１でいえば，数字の１を含むアイコンが一次ウィンドウ，２を含むアイコンが二次ウィンドウ，文字列´MDB´を含むアイコンがモード付きダイアログボックスを表現している．

そして，ウィンドウの表示状態や選択可能状態

およびブロックの状態に変更を与えるような，イベントやコマンド列をフローとした．フローには様々な種類が考えられるが，open,close,implicit closeの３種類のフローの情報をEASELプログラムから生成することにした．各フローは矢印で表現され，色で区別される．

openフローは，あるウィンドウに属するレスポンス文から，かつ別のウィンドウが画面に表示されるような場合に存在する．たとえば，

```
response to item SaveAs from WindowA
    call ProcessDialogBox1()
.....
subroutine ProcessDialogBox1() is
  begin guarded
    response to start
      make DialogBox1 visible
  end
......
```

は，WindowAからDialogBox1にopenフローがあることに対応している．

closeフローは，あるウィンドウに属するレスポンス文において，そのウィンドウ自身を画面から消し，そのレスポンス文が含まれるブロック構造を終了させてしまう場合に存在する．たとえば，

```
response to Cancel in DialogBox2
  make DialogBox2 invisible
  leave block
```

は，DialogBox2からcloseフローが出ている可能性があることに対応している．

さらに，標準的な機能である，取消やヘルプのための操作の存在を調査するために，

a)'Cancel'という文字列を含む押しボタンからによって起きるcloseフロー

b)'HELP'という文字列を含む押しボタンまたはプルダウンメニュウによっておきるopenフロー

といったフローの部分的な表示を検討中である．

5. 視覚化を利用したユーザーインターフェイスの操作構造のレビュー例

図２はSUITで作成可能なアプリケーションの出力画面例である．このアプリケーションは，ファイルの保存(SaveAs)という機能を提供している．一次ウィンドウ(タイトルにExampleと表示されているウィンドウ)に加えて，ダイアログボックス(同じくSave As)，ヘルプウィンドウ(Help Window)，メッセージボックス(Example - Save As)の４個のウィンドウを持っている．

Listing 1は，このアプリケーションをEASELで記述した例である．一見すると，グラフィカルユーザーインターフェイスに必要な操作を十分に提供しているように見えるが，SUITで，openフローやcloseフローを視覚化してみると（図３，４），次のような操作が未定義なことがわかる．

(1)ダイアログボックスからヘルプウィンドウを開く操作．

(2)ダイアログボックス上での取消(Cancel)操作．

また，メッセージボックスで確認が行われたときに，一次ウィンドウに直接戻れないなどの不自然な設計がされていることもわかる．

そこで，SUITの操作構造エディタによって，直接操作でフローの編集を行うと，Listing 2のようなEASELプログラムの修正が行われ，ユーザーインターフェイスが改善される．

６．まとめと今後の予定

SUITはまだ作成中であるが，当初の設計目標とした操作構造の視覚化による直接操作の実現とユーザーインターフェイスの設計支援は，現在の実現方法で達成可能であることが確認できた．EASELプログラムの解析によって生成したノードとフローによって，グラフィカルユーザーインターフェイスの操作構造の実用的な視覚化が可能になった．ノードとフローに対する編集操作によって，直接操作による操作構造の標準的な変更を可能にした．さらに，ノードとフローの情報を利用していくつかユーザーインターフェイスの設計支援が可能になった．

SUITは年内の完成を目指しており，製品レベルの複雑度をもったアプリケーションのプロトタイプ作成に利用する予定である．SUITの実際の効果

については，その結果をもとに報告したい．さらに，操作構造の視覚化だけでなく，プロトタイプの使用記録やタスク分析なども利用して，統合的なグラフィカルユーザーインターフェース評価設計環境として発展させて行きたいと考えている．

7．参考文献

[1]来住，山本，甲：ソフトウェア開発におけるユーザーインターフェイス評価のケース・スタディ，第6回ヒューマンインターフェイスシンポジウム，1990年10月発表予定

[2]IBM Corp.: SAA Common User Access Advanced Interface Design Guide, SC26-4582, 1989

[3]W.M.Newman, R.F.Sproull: Principles of Interactive Computer Graphics, McGraw-Hill, 1979

[4]IBM Corp.: EASEL Version 1.1 for OS/2 Extended Edition, Developer's Guide, SC38-7034, 1990

※OS/2はIBM Corp.の商標です．

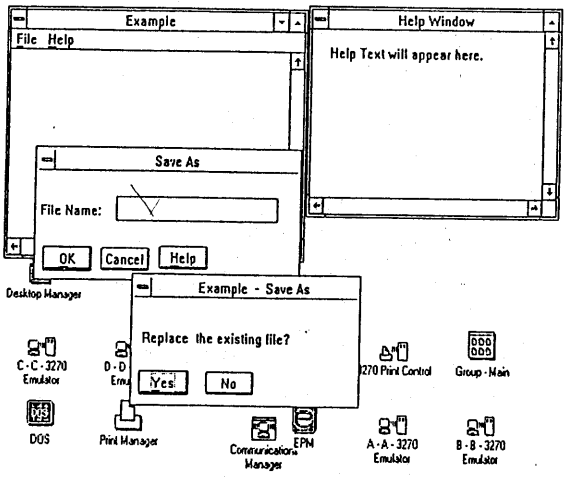

図2．アプリケーション例

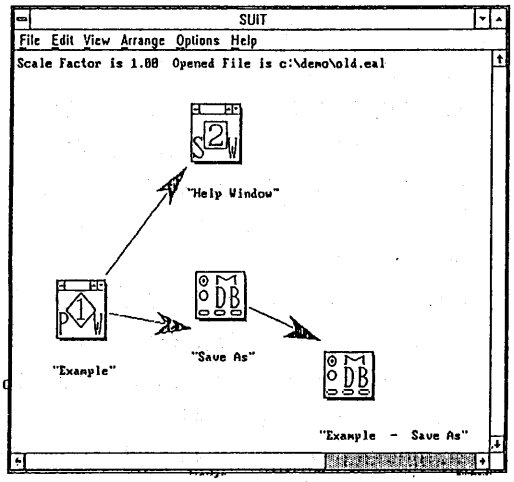

図3．openフローの例

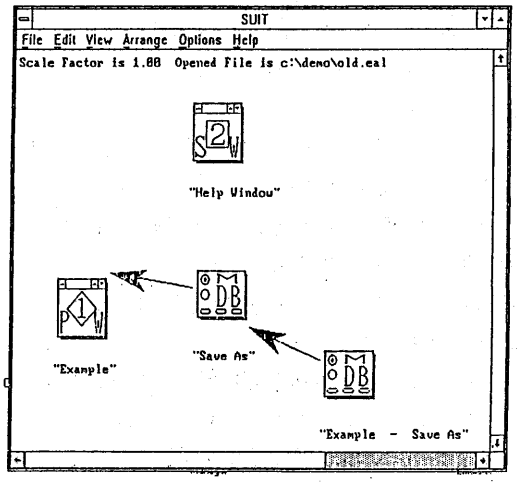

図4．closeフローの例

```
# Listing 1: Example Application in EASEL- Old version
#   Comments start with '#'
#   ..... shows deleted lines.
.....
#
# Action Bar Template Definition(s)
#
action bar PrimaryWindowABCUA is
    pulldown File text "~File"
        choice SaveAs text "Save ~As..."
        separator
        choice Exit text "E~xit"
    end pulldown
    pulldown Help text "~Help"
        choice ExtendedHelp text "~Extended help..."
    end pulldown
end action bar
#
# Primary Window Region Definition
#
enabled visible color 26 primary graphical region PrimaryWindow
    size 318 180
    at position 13 217
    title bar "Example"
    action bar PrimaryWindowABCUA
#
# Secondary Window Region Definition(s)
#
enabled invisible dialog region SecondaryWindow1
    size 176 80
    at position 238 130
    in desktop
    title bar "Help Window"
 ......
#
# Dialog Box Object Definition(s)
#
enabled invisible modal dialog box DialogBox1
    size 196 50
    at position 28 114
    title bar "Save As"
 ......
enabled visible push button Control1
    size 38 12
    at position 94 5
    in DialogBox1
    text "~Help"
 ......
enabled invisible modal dialog box DialogBox2
    size 168 49
    at position 45 75
    title bar "Example  -  Save As"
 ......
boolean variable DidOKCUA DuplicateFile
#
# Subroutine Definition(s)
#
subroutine ProcessDialogBox2( boolean : DidOKCUA ) is
    begin guarded
        response to start
            make DialogBox2 visible
        response to OK in DialogBox2        # A 'close' flow
            copy true to DidOKCUA
            make DialogBox2 invisible
            leave block
        response to Cancel in DialogBox2    # Another 'close' flow
            copy false to DidOKCUA
            make DialogBox2 invisible
            leave block
    end
subroutine ProcessDialogBox1( boolean : DidOKCUA ) is
    begin guarded
        response to start
            make DialogBox1 visible
        response to OK in DialogBox1              # An 'open' flow & a 'close'
            copy true to DidOKCUA                 #  flow start from here .
            # Duplicate file name check has been omitted.
            copy true to DuplicateFile
            if DuplicateFile then
                call ProcessDialogBox2(DidOKCUA) # part of the 'open' flow
            else
                make DialogBox1 invisible         # part of the 'close' flow
                leave block
            end if
        # A 'close' flow and an 'open' flow will be added here.
    end
#
# Response Definition(s)
#
response to item SaveAs from PrimaryWindowABCUA          # An 'open' flow
    call ProcessDialogBox1( DidOKCUA )                   #
response to item ExtendedHelp from PrimaryWindowABCUA    # An 'open' flow
    make SecondaryWindow1 visible                        #
response to item Exit from PrimaryWindowABCUA
    exit
# end of Listing 1
```

```
# Listing 2: Example Application in EASEL- New version
#   Comments start with '#'
#   ..... shows deleted lines.
.....
subroutine ProcessDialogBox2( boolean : DidOKCUA ) is
    begin guarded
        response to start
            make DialogBox2 visible
        response to OK in DialogBox2        # A 'close' flow is
            copy true to DidOKCUA           #  changed to a 'implicit flow'
            make DialogBox2 invisible       #
            leave block                     #
        response to Cancel in DialogBox2    # A 'close' flow
            copy false to DidOKCUA          # starts from here.
            make DialogBox2 invisible       # ##
            leave block                     #
    end

subroutine ProcessDialogBox1( boolean : DidOKCUA ) is
    begin guarded
        response to start
            make DialogBox1 visible
        response to OK in DialogBox1              # An 'open' flow and 'close' flow
            copy true to DidOKCUA                 #    start from here
            # Duplicate file name check has been omitted.
            copy true to DuplicateFile
            if DuplicateFile then
                call ProcessDialogBox2(DidOKCUA) # part of the 'open' flow
                if DidOKCUA then                 # The 'if' statement is added
                    make DialogBox1 invisible    # to create a 'implicit close' flow
                    leave block                  # from DialogBox2 to PrimaryWindow
                end if                           #
            else
                make DialogBox1 invisible        # part of the 'close' flow
                leave block
            end if
        response to Cancel in DialogBox1        # A 'close' flow is added here.
            copy false to DidOKCUA              #
            make DialogBox1 invisible           #
            leave block                         #
        response to Control1 in DialogBox1      # A 'open' flow is added here
            make SecondaryWindow1 visible       #
    end
......
# end of Listing 2
```