修 士 論 文

形式的オブジェクト指向方法論を用いた携帯情報端末の開発実験

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

濱崎 章光

2003 年 3 月

修 士 論 文

形式的オブジェクト指向方法論を用いた携帯情報端末の開発実験

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授
審査委員　二木厚吉 教授
審査委員　権藤克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

110105 濱崎 章光

提出年月: 2003 年 2 月

概 要

オブジェクト指向の形式化/開発環境プロジェクトの開発したプロトタイプ実行 CASE ツール **F-Developer** について、操作法習得上の問題点を検出した。

操作法の習得後、このツール上で現実規模の分析モデルの開発実験を行った。この結果、ツールの有効性を確認した。

目 次

図 目 次

表 目 次

第1章　はじめに

オブジェクト指向開発法は大規模システムの開発に非常に有効であり、実際のシステム開発でも採り入れられている。オブジェクト指向開発法を上流では、分析モデルが構築される。この分析モデルでは、対象とする概念が非常に抽象的なものであり、それゆえ、矛盾や誤りを含みやすいという問題がある。この誤りが開発の下流工程で発見された場合、修復には大きなコストがかかってしまう。また、発見されない場合はシステムに大きな欠陥をもたらすことになる。

分析段階でのモデル構築技術としてUML記法がある。UMLでは複数のモデルを提案しているが、モデル間の意味が曖昧であるため、計算機で支援することができない。そこで、オブジェクト指向技術の形式化/開発環境プロジェクトでは、分析モデル間の意味を形式的に定義したF-Modelを構築した。これにより、計算機支援によるプロトタイプ実行が可能になる。このF-Model技術を利用した、プロトタイプ実行CASEツールF-Developerは既に実装を完了している。本稿ではF-Developerの有用性を確認するため、携帯情報端末の開発実験を行う。

本研究ではF-Developerの操作性を習得することからはじめた。その際、ツール技術習得の障壁がどこにあるかを調査し、分かりやすいマニュアル作成の足掛かりとした。

操作法を習得した後、ツールの実用性を確認した。携帯情報端末モデルの論理的構造と振る舞いを明らかにし、分析モデルを構築し、プロトタイプ実行を行った。このモデルはダイヤルプッシュからはじまる基本的な電話発着機能、各種メニュー表示とその説明機能、簡単なメール機能を持っている。これらの機能の振る舞いをプロトタイプ実行させ、直感的な振る舞いを確認した。

本章の構成を述べる。第2章では既存技術の概要を説明する。第3章ではツール調査に基づく分かりやすいマニュアルを提案し説明する。第4章では携帯情報端末モデルを開発を行ったことの報告をする。

第2章　計算機支援環境 F-Developer

計算機支援環境F-Developerには現在３つのサブシステムが容易されている。検証支援環境(F-Verifier)、モデル構築支援環境(モデルエディタ)、プロトタイプ実行環境(F-Prototyper)である。検証支援環境F-Verifierでは、ML上に実現されている定理証明系HOLを用いて、F-Modelに基づいた検証を行うためのフレームワークを提供する。モデル構築支援環境モデルエディタでは、GUIやMLによるモデル構築インターフェースを提供する。プロトタイプ実行環境F-Prototyperでは、構築したモデルに対するプロトタイプ実行及びテストを行うフレームワークを提供する。
実装は定理証明器HOL、プログラム言語ML、Javaを用いて行っている。HOLは高階述語論理を扱うための定理証明器であり、MLで実装されている。HOLでは、それを構成するML関数を用いたMLプログラムを作成することにより、柔軟に証明支援を行うことができる。

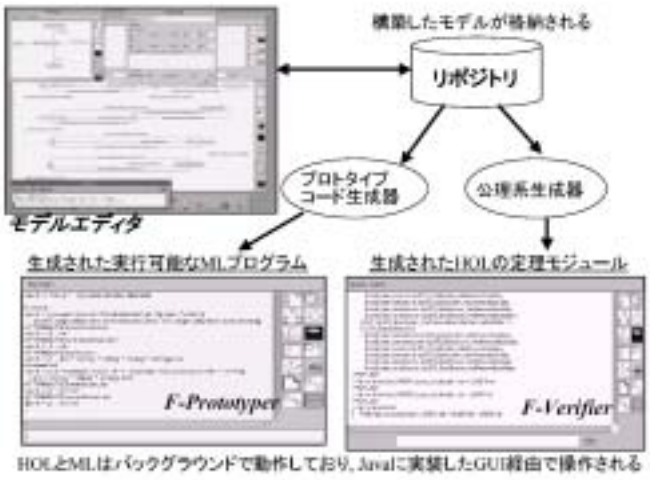

図 2.1: F-Developer

表 2.1: クラス *C1* の属性/関数

属性識別子名	ML 型
a1	int
関数識別子名	ML 関数
countdown	fun countdown(i)=i-1
ifPositive	fun ifPositive(i)=if i> 0 then else true false

2.1 モデルエディタ

モデルエディタとは、オブジェクト指向方法論の形式化/開発環境プロジェクトの提案する形式的オブジェクト指向方法論に基づいた分析モデルを構築するための GUI である。

モデル開発者はモデルエディタ上で UML のクラス図と状態遷移図に似た**基本クラスモデル** (図 2.2) と**基本ステートチャートモデル** (図 2.3) を構築する。
基本クラスモデルを作成する際、に関数と属性にそれぞれプログラミング言語 ML による関数、型を定義する。(表 2.1)
次に基本クラスモデルにおいてクラス図で定義した関数、属性を、アクション式、ガード条件式において ML プログラムの一部として使用する。

図 2.2: 基本クラスモデル

表 2.2 のアクション式 a1:=countdown(a1) に注目する。
ここで基本クラスモデルにおいて定義されている識別子 (a1,countdown,ifPositive) は基本ステートチャートモデルのアクション式において ML プログラムとして結合されている。ここでモデル間に共通の識別子が定義された。この結合された ML プログラムはリポジトリに格納され、構築したモデルの情報とともにプロトタイプ実行環境でデータとして扱うことになる。

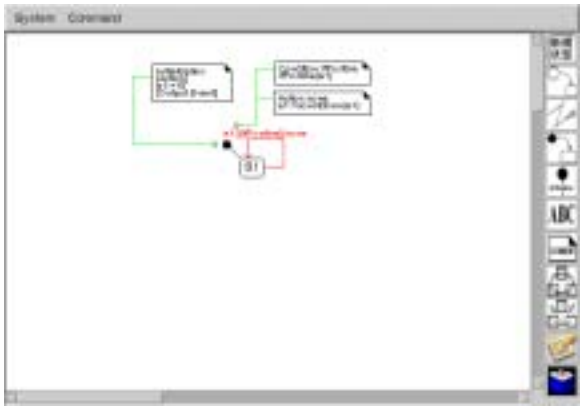

図 2.3: 基本ステートチャートモデル

表 2.2: クラス *C1* の状態遷移

アクション式名	ML 記述
none	a1:=countdown(a1)
ガード条件名	ML 記述
ifPositive	ifPositive(a1)

2.2 F-Prototyper

F-Prototyper では、構築したモデルのプロトタイプ実行を ML 上で行うことができる。ここでは構築したモデルを実行可能な ML プログラムに自動変換する方式をとっている。プロトタイプ実行は、分析モデルの要素に ML の概念を用いて意味を与えることにより、ML インタプリタ上で分析モデルの動作をシミュレートするものである。これは、F-Prototyper により行うことができる。F-Prototyper では、そのプロトタイプコード生成器が、リポジトリの情報を基にして ML の実行可能ソースコードを自動生成する。生成されるソースコードには、オブジェクトを実体化するための関数やオブジェクトの振舞いを定義した関数が含まれている。このソースコードがバックグラウンドで動作している ML インタプリタに読み込まれ、F-Prototyper のウィンドウを用いて対話的にプロトタイプ実行を行うことができる。プロトタイプ実行は、段階的にモデルを構築していくオブジェクト指向開発において非常に有用である。

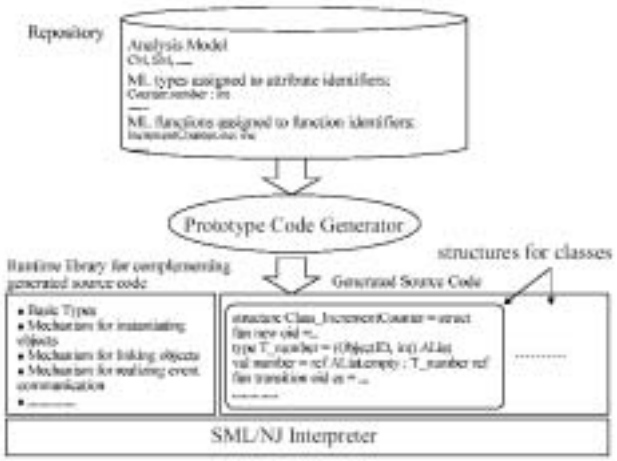

図 2.4: プロトタイプ実行の仕組み

2.2.1 スクリプトエディタ

ML のプログラミングは、プログラムを構成する種々の資源を定義し、それらを高階の関数などを使って組み合わせていくことによって行われる。
プロトタイプ実行環境はバックグラウンドで、オブジェクト指向を表現したいくつかのモジュール (structure 構文) を用意している。
モデル開発者はこのモジュールを組み上げて ML プログラムを構成していかなければならず、どのようにプログラムを自動生成させるか、モジュールを積み上げていく順番（実行系列）を決めなければならない。例えば、ML プログラムでクラスオブジェクトを生成したいとする。クラスオブジェクトの生成にはその主であるクラスの strcture 構文を先にイ

ンタプリタに読み込ませていなければならない。積み上げていくモジュールの設定を行う編集用の GUI がスクリプトエディタである。図 2.5
読み込ませる手順については 3.2.1 章で詳細に述べている。
以下はインタプリタ上で自動生成される ML プログラムである。このプログラムはイベントの structure を表している。

```
- structure Event_e1 :
  sig
    type CurrentStates = (ObjectSystem.ObjectID,DynamicModel.StateID) AList
    type InternalObjects =
      (ObjectSystem.ObjectID,ObjectSystem.ObjectID Set) AList
    type LinkConf =
      (ObjectSystem.LinkID,ObjectSystem.ObjectID * ObjectSystem.ObjectID) AList
    type OutputEvents =
      (ObjectSystem.ObjectID,
       {event:ObjectSystem.EventInstanceID, link:ObjectSystem.LinkID} Bag)
        AList
    exception notFoundEvent
    exception runtimeError of string
    val create_eventinstances : DynamicModel.EventID
                                -> int -> ObjectSystem.EventInstanceID list
    val eval_ineventexp : DynamicModel.InEventExp
                          -> ObjectSystem.EventInstanceID Set -> bool
    val getEventOfLink : {event:ObjectSystem.EventInstanceID,
                          link:ObjectSystem.LinkID} list
                         -> ObjectSystem.LinkID
                            -> ObjectSystem.EventInstanceID list
    val get_evtins : DynamicModel.EventID
                     -> ObjectSystem.EventInstanceID list
                        -> ObjectSystem.EventInstanceID
    val lookup_linkOFobject : (''a * (''b * ''b)) list -> ''b -> ''a Set
    val message : string -> OS.Process.status
    val removeEventOfLink : {event:ObjectSystem.EventInstanceID,
                             link:ObjectSystem.LinkID} list
                            -> ObjectSystem.LinkID
                               -> {event:ObjectSystem.EventInstanceID,
                                   link:ObjectSystem.LinkID} list
    val warning : string -> unit
  end
```

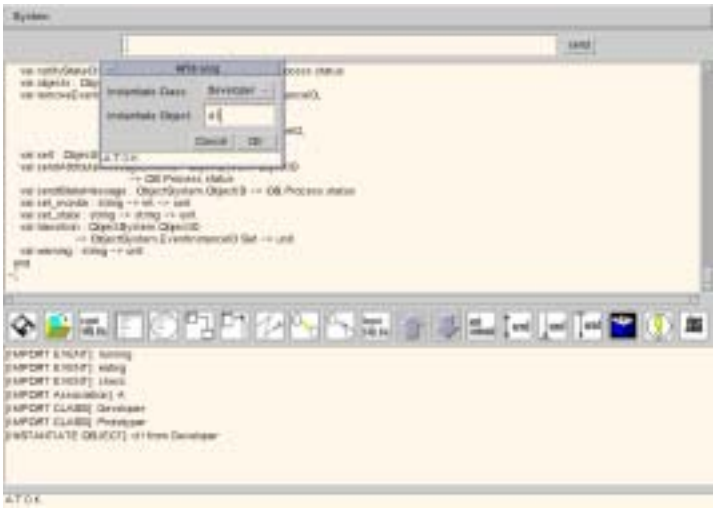

図 2.5: スクリプトエディタ

2.2.2 アニメータ

構築したモデルの状態遷移を確認できる GUI である。
SML インタープリタで構築した ML プログラムの情報は JNI(Java Native Interface) を経由して、Java で実装したアニメータに渡し、アニメーターはその情報を GUI に表示する。
図 2.6

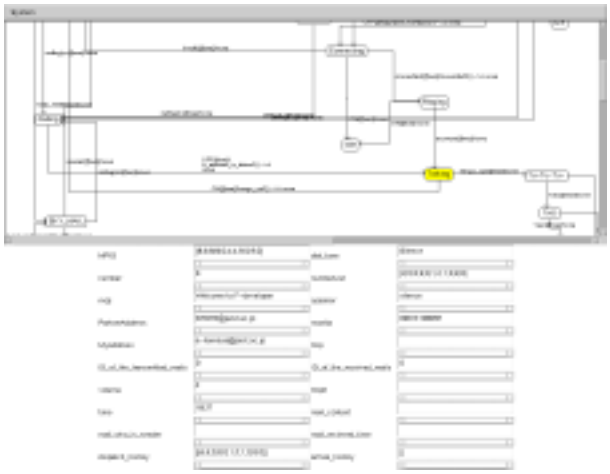

図 2.6: アニメーション

第3章　F-Developerの操作法

計算機支援環境 F-Developer について調査を行い、その操作法を習得した。この章では F-Developer の操作法をもって調査の報告を行う。尚、F-Developer の持つ 3 つのサブシステムのうち、検証支援環境 F-Verifier には今回触れていない。

図 3.1 は F-Developer の起動画面である。

図 3.1: F-Developer 起動画面

図 3.1 では、アイコン左から順に以下のことができる。

- model ファイルの保存
- model ファイルのロード
- モデルエディタの表示 (図 3.2)
- F-PrototyperGUI の表示 (図 3.3)
- F-VerifierGUI の表示 (図 3.4)
- システムの終了

3.1　モデルエディタ

モデルエディタ (図 3.2) では F-Model を構築する。図 3.2 では、アイコン上から順に以下のことが行える。

- クラスを作成する
- 関連を作成する

図 3.2: モデルエディタ

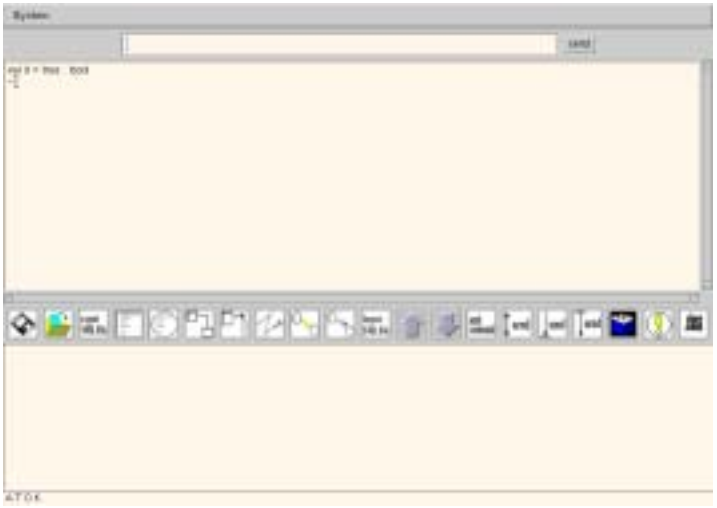

図 3.3: F-Prototyper

図 3.4: F-Verifier

- 集約を作成する
- 汎化を作成する
- 状態遷移図を作成する
- 下線付見出しを作成する
- 矢印付メモを作成する
- GUI アイコンのグループ化を行う
- GUI アイコンのグループ化解除を行う
- 各種プロパティを定義付ける
- 作成したモデル等を削除する

3.1.1 基本クラスモデルの構築

新規クラス作成　上から 1 番目のアイコンでクラスを作成できる。
クリックすると ClassName 記入ダイヤログが表示され、クラス名を記入し OK をクリックすると、モデルアイコンが表示される (図 3.5)

図 3.5: 新規クラス作成

関連作成　上から 2 番目のアイコンでクラスの関連を作成できる。
表示されるダイヤログで操作のキャンセルが可能である。(図 3.6)
尚、この操作のあと関連のプロパティシートに詳細を書き込む必要がある。

集約作成　上から 3 番目のアイコンでクラス間の集約を作成できる。
表示されるダイヤログで操作のキャンセルが可能である。(図 3.7)

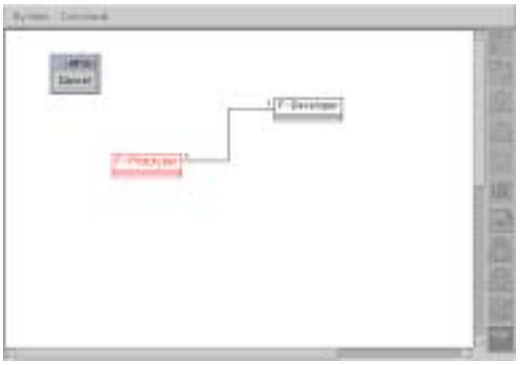

図 3.6: 関連作成

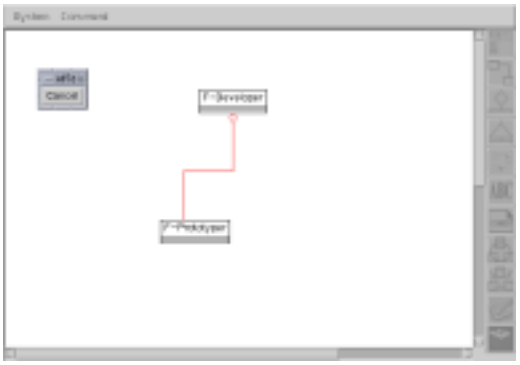

図 3.7: 集約作成

汎化作成　3.2 上から 4 番目のアイコンでクラス間の汎化を作成できる。
表示されるダイヤログで操作のキャンセルが可能である。(図 3.8)

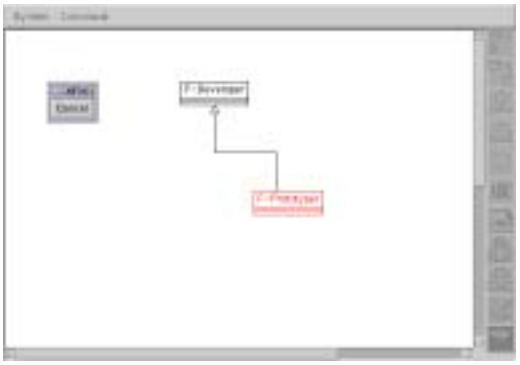

図 3.8: 汎化作成

関連のプロパティシート　作成したモデルに定義付けを行いたい場合、または修正したい場合は、モデルを選択し、下から 2 番目のアイコンをクリックする。(図 3.9)
ここでシステム特有の記法があり、多重度 ..* を表したい場合、プロパティシートの Max のボックスに-1 を記入する。(図 3.9)
また関連の属性を定義したい場合、Link Propaty の add をクリックすると図 3.10 があらわれ、ここに記入する。

図 3.9: 関連のプロパティシート

状態遷移図　モデル上のクラスを選択し上から 5 つ目のアイコンをクリックするとそのクラスの状態遷移図を作成できる新たな GUI が表示される。(3.11)
１つのクラスに１つの状態遷移図 GUI を割当てモデルを構築していく。
詳細は次のセクションで紹介する。

見出しの作成　上から 6 番目のアイコンで「下線付見出し」を記述できる。図 (3.12)

図 3.10: 関連/属性のプロパティシート

図 3.11: 状態遷移図の GUI

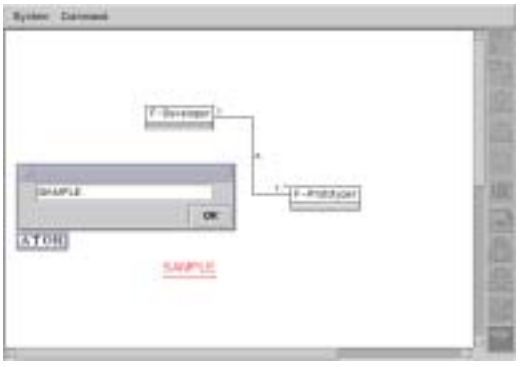

図 3.12: 見出しの作成

コメント文の作成　上から 7 番目のアイコンで「矢印付のコメント文」を記述できる。図 (3.13)

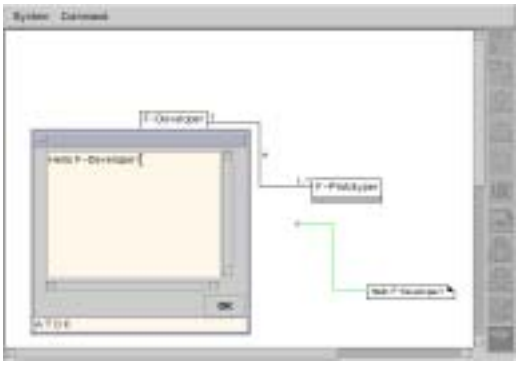

図 3.13: コメント文の作成

モデルのグループ化と解除　上から 8 番目/9 番目のアイコンで選択したモデルをグループ化/解除できる。
図 3.13 では 2 つのクラスと 1 つ関連をグループ化し GUI 上を移動させている。

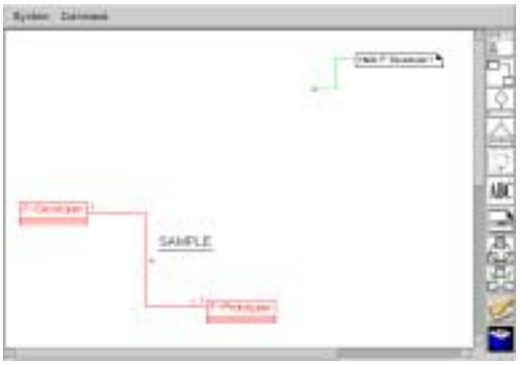

図 3.14: モデルのグループ化と解除

クラスのプロパティシート　クラスに属性などの定義付けを行いたい場合、GUI 上のクラスを選択し、下から 2 番目のアイコンをクリックする。(図 3.15)
このプロパティシートで属性には ML の型を、関数には ML の関数プログラムをそれぞれ記述する。Attribute で add をクリックすると図 3.16 のダイヤログが表示される。
Name:のボックスには属性名を記述、ML Type:のボックスには ML の型、HOL Type:のボックスには HOL の型を記述する。HOL のよる記述は検証支援環境 F-Verifier で利用する。

Description:のボックスにはメモなど自然言語を記述してよい。
Function でを add クリックするとの図 3.17 ダイヤログが表示される。
Name:のボックスには関数名を記述、ML Description:のボックスには ML の関数プログラムを記述する。

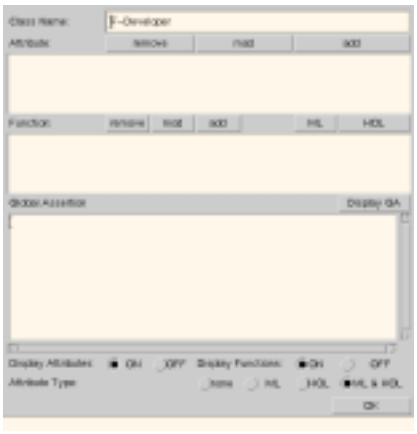

図 3.15: クラスのプロパティシート

図 3.16: 属性のプロパティシート

3.1.2 基本ステートチャートモデルの構築

クラス図 GUI 上でクラスを選択し上から 5 つ目のアイコンをクリックすると状態遷移図を描ける GUI があらわれる。(図 3.11)
1 つのクラスに 1 つの状態遷移図を描いていく。

状態作成 上から 1 番目のアイコンをクリックすると、ダイヤログボックスがあらわれる。(図 3.18)
State Name:のボックスに状態名を書くと GUI に状態アイコンが表示される。(図 3.18)

図 3.17: 関数のプロパティシート

図 3.18: 状態の新規作成

ラインの表示　上から 2 番目のアイコンで GUI 上のモデル間にラインを引くことができる。表示されるダイヤログをクリックすると操作はキャンセルされる。(図 3.19)
このラインはイベントのプロパティシートによって定義されなければならない。

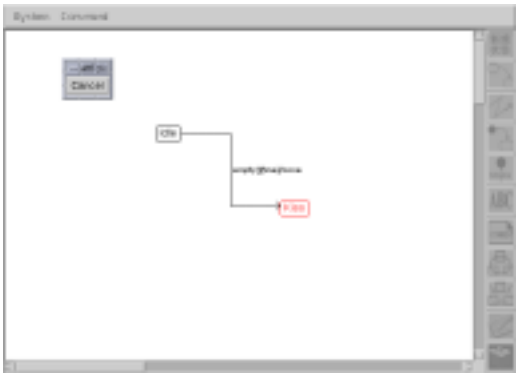

図 3.19: ラインの表示

入出力イベント作成　上から 3 番目のアイコンで入力イベント、出力イベントを作成する。(図 3.20)
Input Event:で new をクリックすると図 3.21 ダイヤログが表示される。
Name: でイベント名を記述し、Attribute: では引数を記述し、ML の型定義する。
Dexcription: にはメモとして自然言語を記述してよい。
Input Event: import で他のクラスで定義されているイベントを読み込むことができる。(図 3.22)

図 3.20: 入出力イベントの作成ダイヤログ

初期状態の決定　上から 4 番目のアイコンでモデルの初期状態を決定する。(図 3.23)

初期状態のプロパティシート　上から 5 番目のアイコンで初期状態を定義する。(図 3.24)
Action: の edit でアクション式を編集し (図 3.25)、 display をクリックすると GUI 上に

図 3.21: イベントのプロパティシート

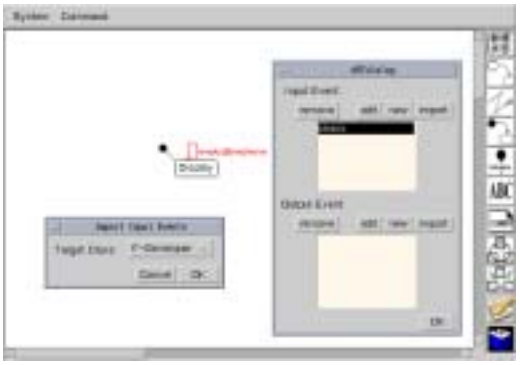

図 3.22: イベントのインポート

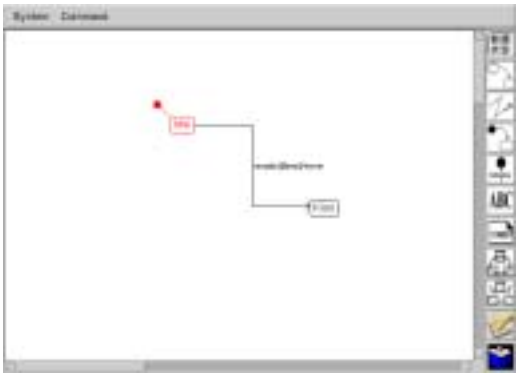

図 3.23: 初期状態の決定

アクション式が表示される。(図 3.24)
図 3.25 では Name: に初期状態の名前、Action Expression: には ML プログラム、 Description: にはメモとして自然言語を記述できる。

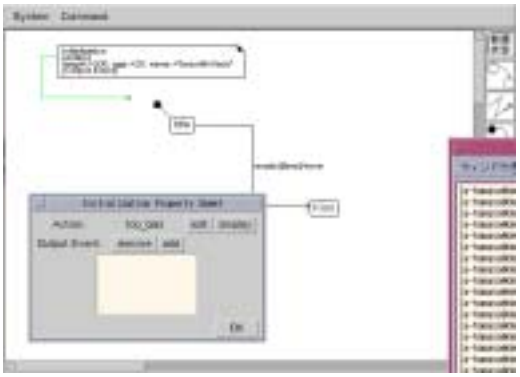

図 3.24: 初期状態のプロパティシート

図 3.25: アクションのプロパティシート

見出しの作成　上から 6 番目のアイコンで「下線付見出し」を記述できる。モデルの規模が大きくなってくると、状態遷移図の機能も増えてくる。F-Developer では 1 つの GUI 上に複数の状態遷移図を書くことができる。機能別にそれぞれ見出しを付けておくとよい。(図 3.55)

コメント文の作成　上から 7 番目のアイコンで「矢印付のメモ」を記述できる。自然言語で記述できる。(図 3.26)

GUI アイコンのグループ化と解除　上から 8 番目/9 番目のアイコンでモデルのグループ化/解除ができる。

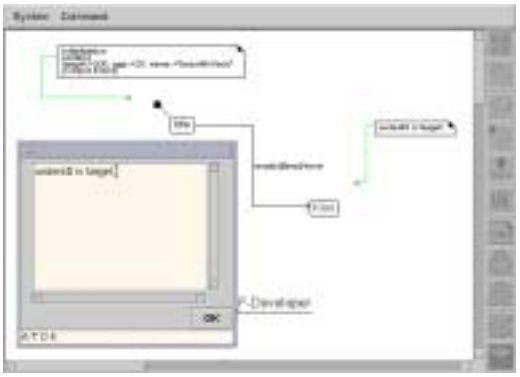

図 3.26: コメント文の作成

遷移のプロパティシート　GUI でラインを選択し、上から 8 番目のアイコンで遷移を定義付する。(図 3.27)
アクションは Action:edit で編集し、ガード条件は Condition:edit で編集し、それぞれ display で GUI に表示させる。
図 3.28 は Action:の edit で表示される。
Action Expression:に ML プログラムを記述する。Description: にはメモとして自然言語を記述できる。
図 3.29 は Output Event: の add で表示される出力イベント生成のプロパティシートである。出力先のリンクや引数等を定義する。
display で表示したアイコンからもプロパティシート (図 3.30) を表記できる。
ここで F-Developer 特有の記法がある。イベントの引数をアクション式で用いる場合、**#イベント名.引数名**と記述する。(図 3.30)

図 3.27: 遷移のプロパティシート

3.2　プロトタイプ実行環境 F-Prototyper

構築したモデルのプロトタイプ実行を行う。
(例題モデル（クラス図 (3.31)、状態遷移図 (3.32),(3.33)）)

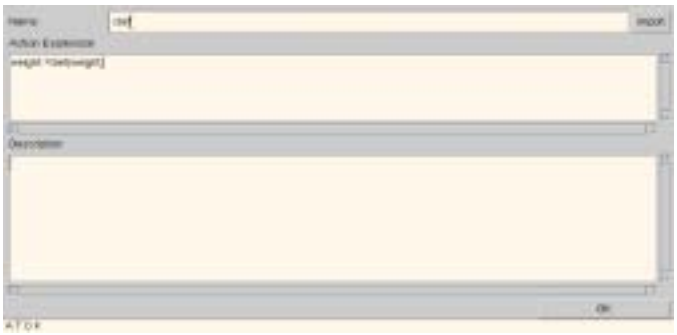

図 3.28: アクション式のプロパティシート 1

図 3.29: 出力イベントのプロパティシート

図 3.30: アクション式のプロパティシート 2

表 3.1 で図 3.3 のアイコンを左から順にその役割を説明した。

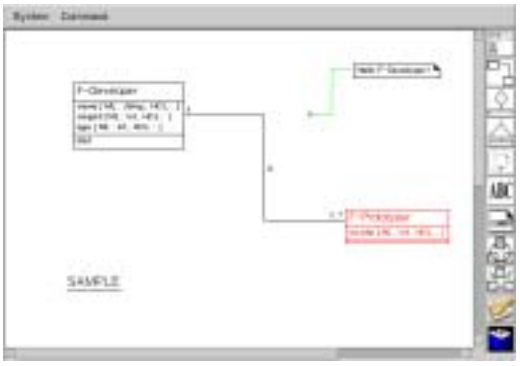

図 3.31: 例題モデルのクラス図

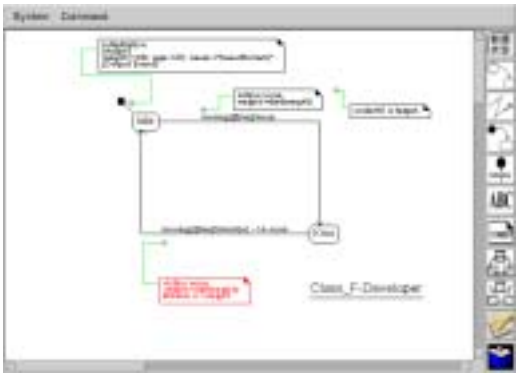

図 3.32: 例題モデルの状態遷移図 1

3.2.1 実行系列の生成

プロトタイプ実行環境では、SML インタプリタ上に ML プログラムを自動生成している。
モデル開発者はまず、SML インタプリタに読み込ませる実行系列を構築しなければばらない。実行系列にはオブジェクトを生成する以前にクラス structure を作らなければならない等、一定の順番規則ある。

IMPORT EVENT SML インタプリタに読み込ませる実行系列には順番規則があり、はじめに読み込ませなければならないのは、イベントの structure である。

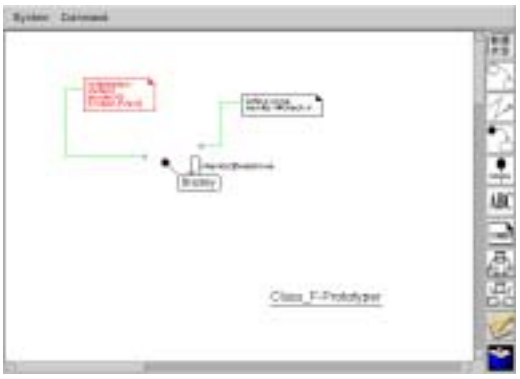

図 3.33: 例題モデルの状態遷移図 2

表 3.1: F-PrototyperGUI の各種アイコンの役割

アイコン	役割
フロッピーディスク	構築した実行系列を coms ファイルに保存する
フォルダ	coms ファイルを開く
export SML File	ML プログラムを sml ファイルに保存する
クラス生成	作成したクラス structure を命令文に加える
イベント生成	作成したイベント structure を命令文に加える
リンク生成	作成したリンク structure を命令文に加える
クラスインスタンス生成	オブジェクトを生成する
イベント送信	イベントを送る
リンクインスタンス生成	オブジェクト間にリンクを生成する
同期送信	同期イベントを送る
import SML File	ML プログラム（sml ファイル）を命令文に加える
矢印上	選択した命令文の順番を前に 1 つずらす
矢印下	選択した命令文の順番を後に 1 つずらす
send comand1	選択した命令文のみ ML インタプリタに読み込ませる
send comand2	表示した全ての命令文を ML インタプリタに読み込ませる
send comand3	選択した命令文以前を ML インタプリタに読み込ませる
send comand4	選択した命令文以降を ML インタプリタに読み込ませる
ゴミ箱	選択した命令文を消去する
ビックリマーク	アニメーション GUI を表示する
class ローダー	不明

左から 5 番目のアイコンでダイヤログを表示させ、読み込ませたいイベントを選択し、スクリプトエディタに並べる。(図 3.34)

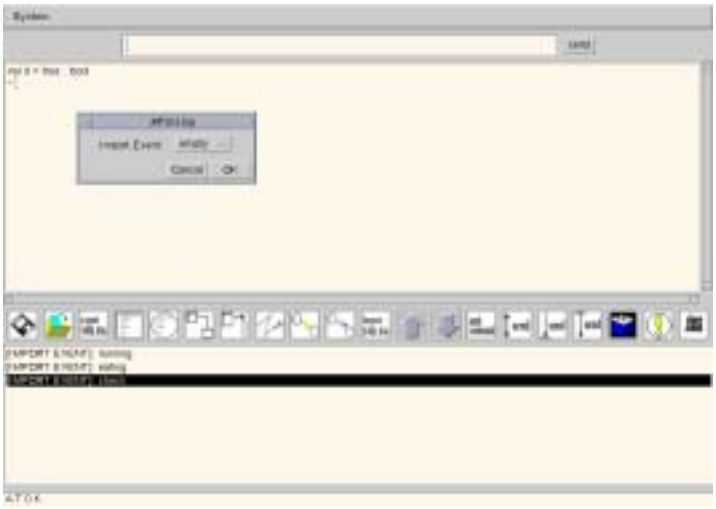

図 3.34: IMPORT EVENT

IMPORT Association　イベントの次に読み込ませなければならないのは関連の structure である。
左から 6 番目のアイコンでダイヤログを表示させ、読み込ませたい関連を選択し、スクリプトエディタに並べる。(図 3.35)

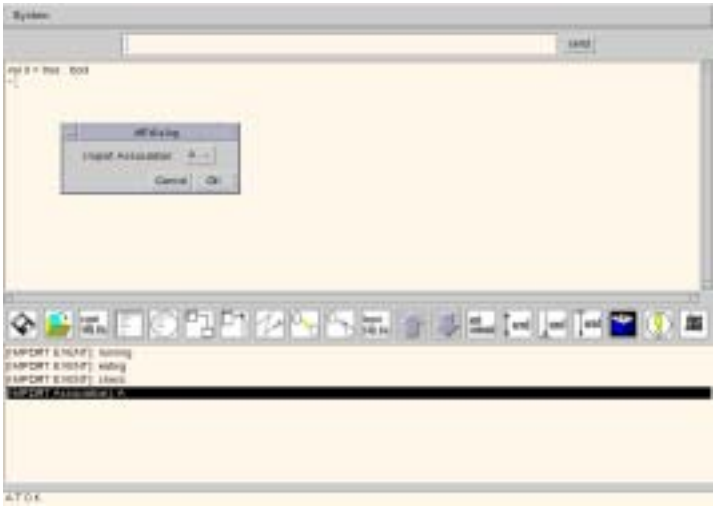

図 3.35: IMPORT Association

IMPORT CLASS　次に読み込ませなければならないのはクラスの structure である。左から 4 番目のアイコンでダイヤログを表示させ読み込ませたいクラスを選択し、スクリプトエディタに並べる。(図 3.36)

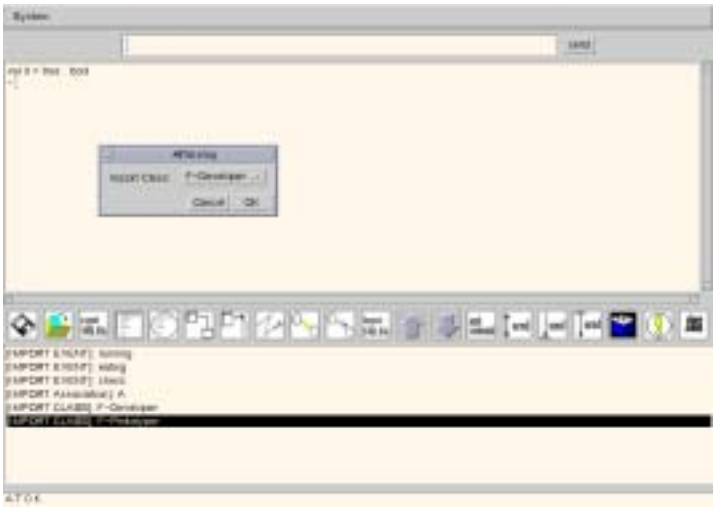

図 3.36: IMPORT CLASS

send command　並べた実行系列のコマンドを実際に SML インタプリタに読み込ませる。分析モデルに記述上の問題点があれば、インタプリタにエラーが表示される。
実行系列をインタプリタに送信するアイコンは 4 つあり、それぞれ (表 3.1) に記す。
図 3.37 は実行系列の上から順に IMPORT Association コマンドまで、右から 5 番目のアイコンを使ってインタプリタに送信した図である。計算機によって ML プログラムが自動生成されているのがわかる。通常モデル開発者は自動生成された ML プログラムを全て管理する必要がないため、インタプリタには必要最低限の ML プログラムしか表示されていない。
自動生成された ML プログラムを確認するためには、GUI 左上の System :EcoBackOn で表示させるか、もしくは SML ファイルとして保存し確認する。
インタプリタ上で出力の最後に

```
..
end
-
```

と表示されていることがわかる。イベントと関連にはモデルに記述上の問題点がなかったことが確認された。(39)

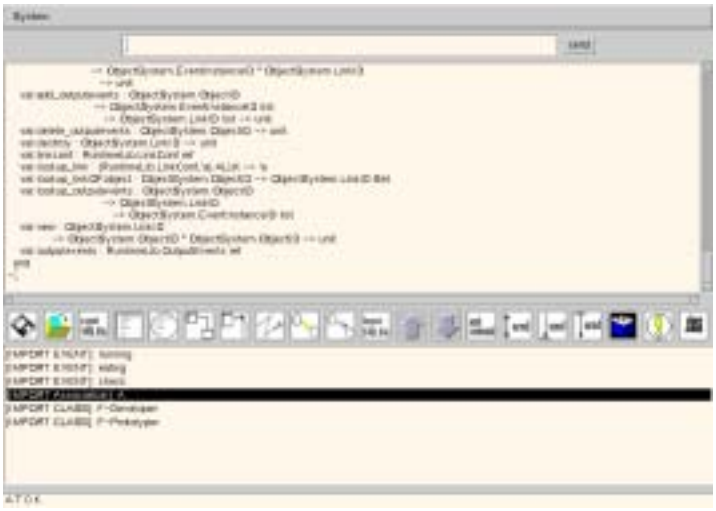

図 3.37: 計算機が ML プログラムを自動生成する

表記法上のエラー　次に実行系列上の

```
[IMPORT CLASS]:F-Developer
```

コマンドを、右から 7 番目のコマンドを使ってインタプリタに読み込ませてみた。ここでエラーが確認された。(図 3.38)

```
[IMPORT CLASS]:F-Prototyper
```

コマンドを送ってみたが同様のエラーの結果が出力され、プロトタイプモデルに表記法上の問題（Syntax error）があることが確認できた。

再度モデルエディタを用いて、クラス名の表記に問題点を確認し、モデルを再構築した。(図 3.39)

インタプリタをリブートさせ (図 3.40)、実行系列から、[IMPORT CLASS]:F-Developer コマンドと [IMPORT CLASS]:F-Prototyper コマンドをゴミ箱を用いて消去し、右から 4 番目のアイコンより新しくクラスコマンドを作った。
右から 6 番目のアイコンを使い実行系列コマンドを一度に読み込ませたが、結果、インタープリタにエラーは確認されず、インタプリタ上に正常に structure を完成させた。(図 3.41)

3.2.2　振る舞いの確認

INSTANTIATE OBJECT　インタプリタ上に structure を生成させた後、その structure を基にオブジェクトを生成させる。右から 7 番目のアイコンをクリックしダイヤログ

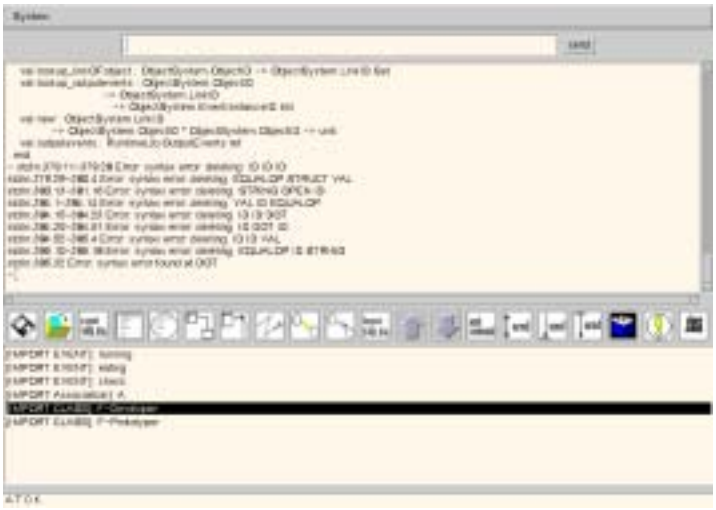

図 3.38: エラーの表示 1

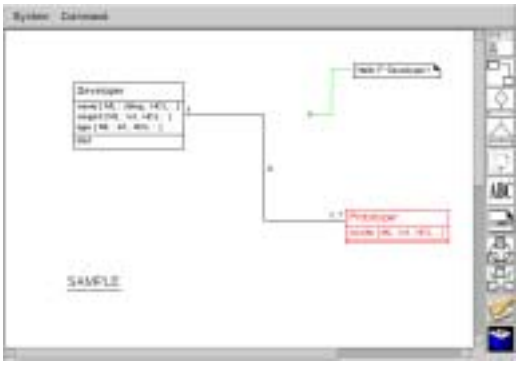

図 3.39: 例題モデルの再構築

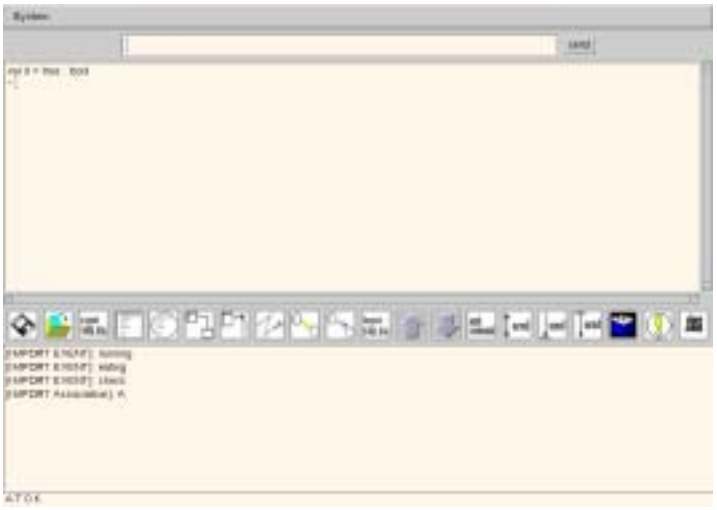

図 3.40: SML インタプリタのリブート

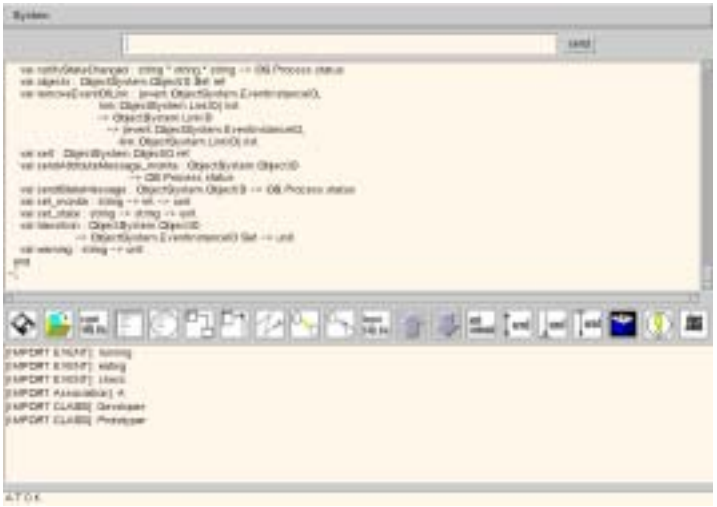

図 3.41: structure 構文の完成

を表示させ、ボックスにオブジェクト名を記述する。(図 3.42)
次に実行系列からコマンドをインタプリタに読み込ませた。

```
val it =():unit
```

としか出力されないがバックグラウンドでオブジェクトは自動生成されている。バックグラウンドは左上の System :EcoBackOn で表示することができる。もしくは SML ファイルとして保存し確認することができる。

Developer クラスから *d1* オブジェクトを生成している様子が確認できる。(図 3.43)
同様に *Prototyper* クラスから *p1* オブジェクトも生成させる。

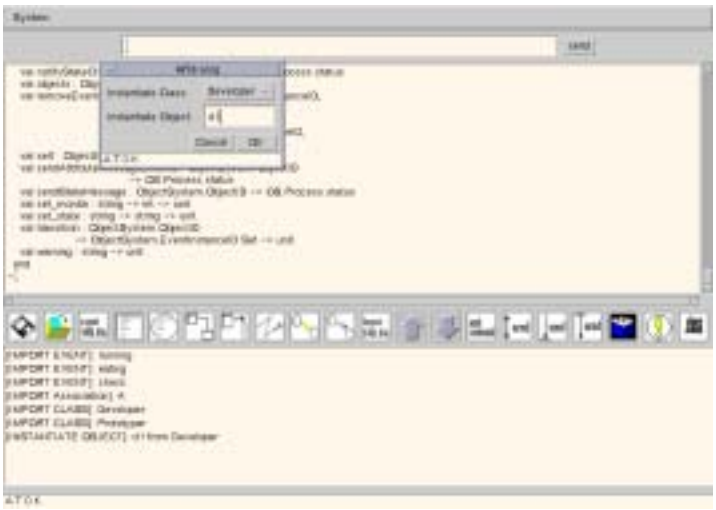

図 3.42: オブジェクト生成のための実行系列生成

ADD LINK　クラスオブジェクトを生成させた後、そのオブジェクト間にリンクを生成させる。右から 7 番目のアイコンをクリックしダイヤログを表示させ、ボックスにリンク名を記述する。(図 3.44)
送信ボタンで [ADD LINK] コマンドをインタプリタに送信する。

アニメーション GUI　オブジェクトやリンクを生成させた後、アニメーションでオブジェクトの振る舞いを確認できる。右から 2 番目のアイコンでダイヤログを表示させ、生成したオブジェクトのアニメーション GUI を表示させる。(図 3.45),(3.46)

SEND EVENT　[IMPORT EVENT] コマンドで生成したイベントを実際に送信する。右から 8 番目のアイコンでダイヤログを表示させ、送信するイベントを選択すし、実行系

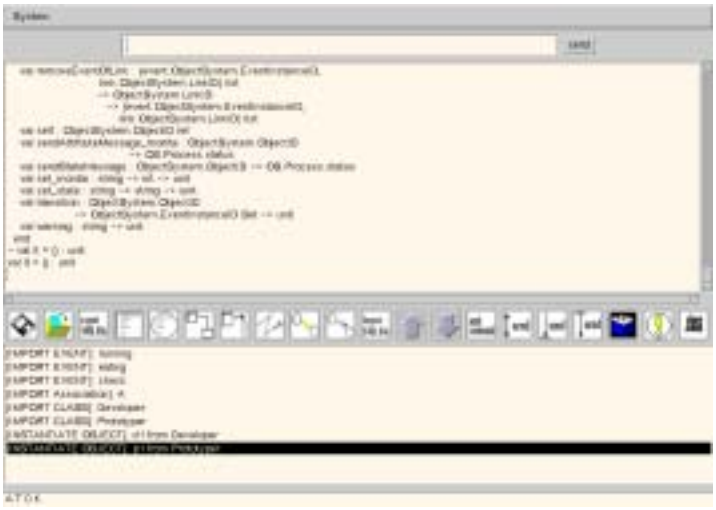

図 3.43: オブジェクトの生成

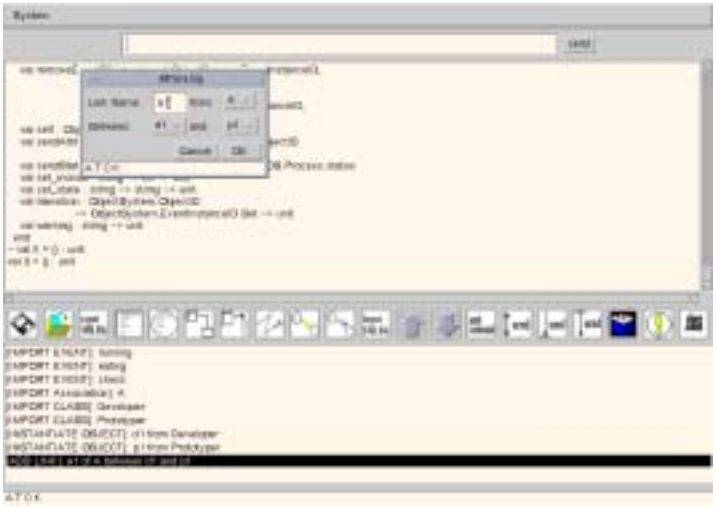

図 3.44: リンクインスタンスの生成

図 3.45: アニメーション 1

図 3.46: アニメーション 2

列上に並べる。
図 3.47 では running() イベントを実行系列上に生成している。
図 3.48 では生成した running() コマンドをインタプリタに読み込ませている。

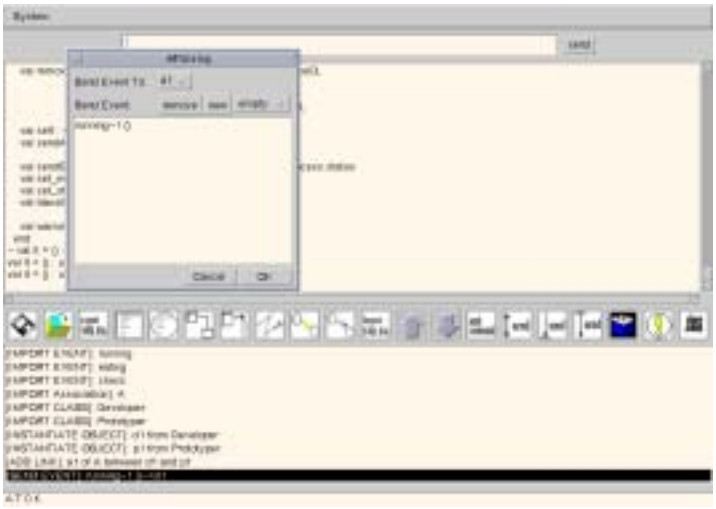

図 3.47: SEND EVENT 実行系列の生成

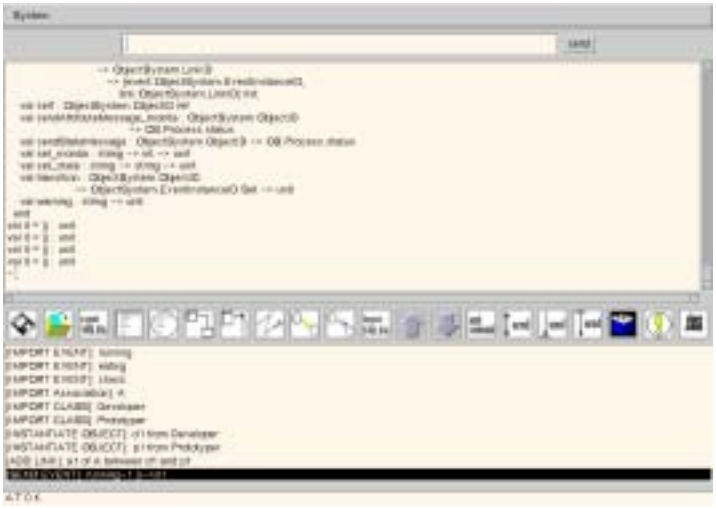

図 3.48: SEND EVENT 実行系列の送信

この時、*d1* オブジェクトのアニメーション GUI が変化する。初期状態の Idle から状態 KISS に状態が遷移し、属性ボックスにも変化が見られ、期待していた通りの振る舞いを確認できた。(図 3.49)

図 3.49: アニメーションによる遷移の確認 1

SYNCHRONAIZATION　イベントの同期をアニメーションで確認するために SYNCHRONAIZATION コマンドを生成する。
右から 10 番目のアイコンよりダイヤログを開き、どのリンクを使ってどのオブジェクトからどのオブジェクトへ同期させるのか決定する。
まずは入力イベントを読み込ませる。例題では図 3.49 の状態より、さらにもう一度、running イベントを送信させ、図 3.50 の遷移を行わせる。ここで SYNCHRONAIZATION コ

図 3.50: アニメーションによる遷移の確認 2

マンドを送信させる。出力イベントはリンク a1 を用いて d1 から p1 へ送信させる。(図 3.51)
p1 オブジェクトのアニメーション GUI の属性 monita に変化が見られ、プロトタイプ実行は期待していた通りの振る舞いを確認できた。(図 3.51)

3.2.3　補助機能

ここではプロトタイプ実行を補助する機能について紹介する。

- SML インタプリタをリブートさせることや、インタプリタの画面上をクリーンにす

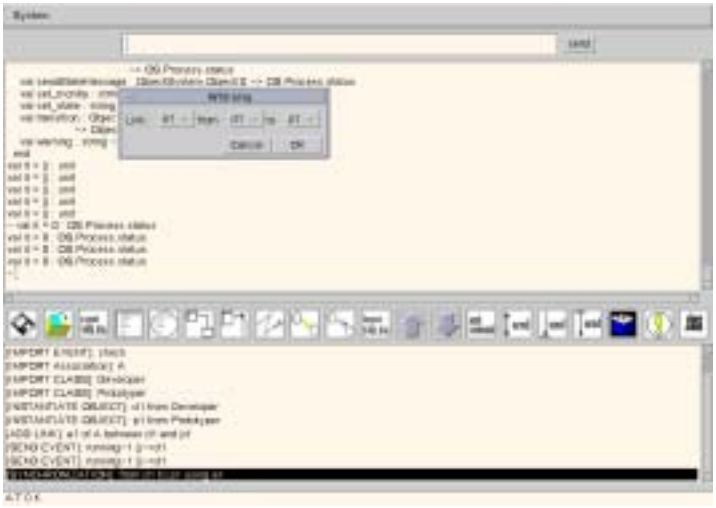

図 3.51: SYNCHRONAIZATION コマンドの生成

図 3.52: アニメーションによる遷移の確認 3

ることができる。

- 通常開発者には直接知る必要がないインタプリタに読み込ませている structure 文を表示させることができる。
- 通常開発者には直接知る必要がないインタプリタに読み込ませている structure 文を表示させないことができる。
- 直接インタプリタに SML コードを書くことも可能である。

スクリプトエディタではコマンドの実行系列を coms ファイルとして保存することができる。(図 3.53)

図 3.53: 実行系列ファイルの保存

スクリプトエディタではコマンドの実行系列を coms ファイルとして読み込むことができる。(図 3.54)
インタプリタによって構築された SML プログラムは sml ファイルとして保存することができる。(図 3.55)

インタプリタによって構築された SML プログラムは sml ファイルとして読み込むことができる。(図 3.56)

モデル開発者は ML プログラムのソースコードを F-Prototyper 上で直接生成することが可能である。(図 3.57)
図 3.57 では入力として、

```
Class_Prototyper.monita\end;
```

図 3.54: 実行系列ファイルの読み込み

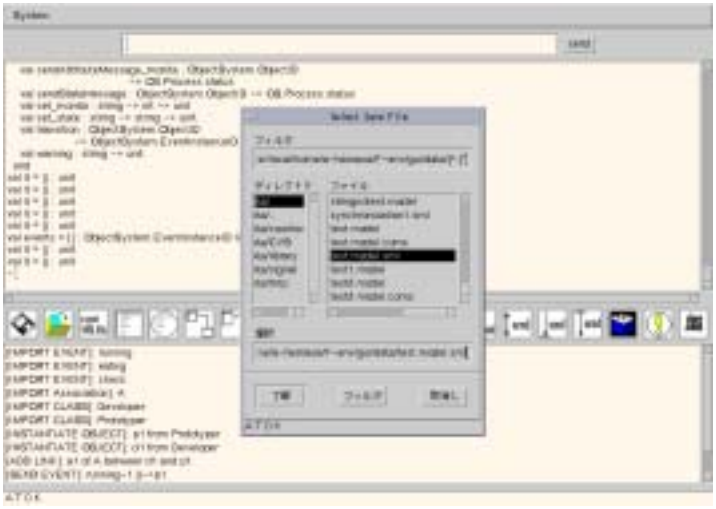

図 3.55: インタプリタ上の ML プログラムを sml ファイルに保存

図 3.56: ML プログラムの読み込み

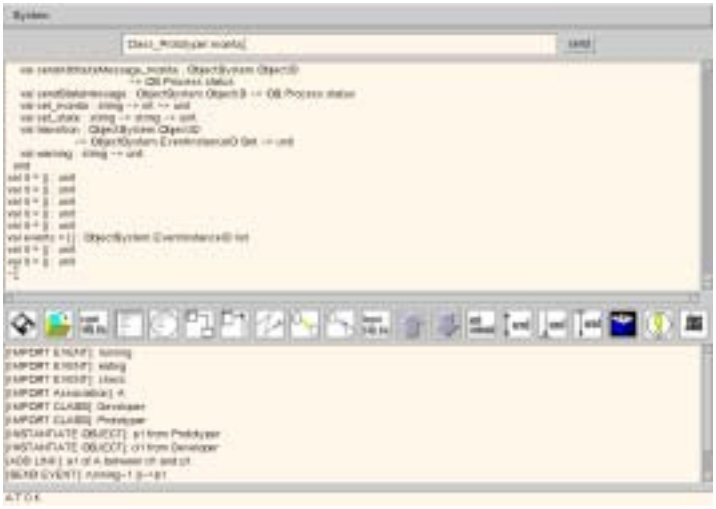

図 3.57: 開発者による ML プログラムの作成

を記述している。このプログラムはPrototyperクラスの属性monitaの値を調査を行う。

```
- Class_Prototyper.monita;
val it = ref[("p1,0")]:Class_Prototyper.Type_monita ref
```

ここで出力

```
:Class_Prototyper.Type_monita ref
```

は

```
ref[("p1,0")]
```

に対してMLシステムが推論した型情報である。(図3.58)
出力の意味は、Prototyperクラスから生成されているオブジェクトはp1であり、そのオブジェクトの属性monitaの値は現在0であることを意味している。
このようにML言語の特徴である対話プログラムが可能である。

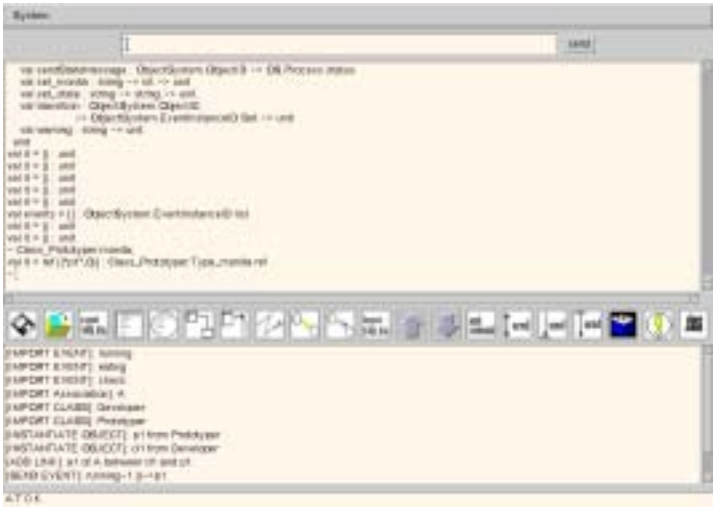

図 3.58: MLプログラムによる対話

3.2.4 ml プログラムによるオブジェクトの実行

new関数　F-Prototyperでは状態遷移図において、あるイベントのアクションとして、オブジェクトを生成させる場合、分析モデル上にオブジェクトを生成させるMLプログラムを記述することで可能となる。
以下のMLプログラムはインタプリタにいくつかのイベントを送った結果、出力されたものである。ここで意味を推測することは容易である。

```
...
.....
(*[INSTANTIATE OBJECT]: c2 from C2*)
Class_C2.new "c2"
 handle RuntimeLib.runtimeError(m) =>
(print ("\n"^m^"\n");raise RuntimeLib.runtimeError(m));
(*[ADD LINK]: a1 of A1 between c1 and c2*)
Assoc_A1.new "a1" ("c1","c2")
 handle RuntimeLib.runtimeError(m) =>
 (print ("\n"^m^"\n");raise RuntimeLib.runtimeError(m));
(*[SEND EVENT]: e1-1 ()->c1*)
Class_C1.transition "c1" [{event = "e1",id = ~1}];

(*[SYNCHRONIZATION]: from c1 to c2 using a1*)
val events = Assoc_A1.lookup_outputevents "c1" "a1";
Class_C2.transition "c2" events;
Assoc_A1.delete_outputevents "c1";
```

例えば、C2 クラスから c2 オブジェクトを生成しているプログラムは

```
Class_C2.new "c2"
```

である。動機を表しているプログラムは

```
val events = Assoc_A1.lookup_outputevents "c1" "a1";
Class_C2.transition "c2" events;
Assoc_A1.delete_outputevents "c1";
```

である。

次にクラスを生成している structure を表示してみると、

```
- structure Class_C1 :
  sig
    type CurrentStates = (ObjectSystem.ObjectID,DynamicModel.StateID) AList
    type InternalObjects =
      (ObjectSystem.ObjectID,ObjectSystem.ObjectID Set) AList
    type LinkConf =
      (ObjectSystem.LinkID,ObjectSystem.ObjectID * ObjectSystem.ObjectID) AList
    type OutputEvents =
      (ObjectSystem.ObjectID,
```

```
   {event:ObjectSystem.EventInstanceID, link:ObjectSystem.LinkID} Bag)
    AList
type Type_a1 = (ObjectSystem.ObjectID,int) AList
exception notFoundEvent
exception runtimeError of string
val a1 : Type_a1 ref
val add_internalobject : ObjectSystem.ObjectID
                         -> ObjectSystem.ObjectID -> unit
val add_object : ObjectSystem.ObjectID -> unit
val class : string
val countdown : int -> int
val create_eventinstances : DynamicModel.EventID
                            -> int -> ObjectSystem.EventInstanceID list
val currentstates : CurrentStates ref
val delete_a1 : ObjectSystem.ObjectID -> unit
val delete_internalobject : ObjectSystem.ObjectID
                            -> ObjectSystem.ObjectID -> unit
val delete_object : ObjectSystem.ObjectID -> unit
val eval_ineventexp : DynamicModel.InEventExp
                      -> ObjectSystem.EventInstanceID Set -> bool
val getEventOfLink : {event:ObjectSystem.EventInstanceID,
                      link:ObjectSystem.LinkID} list
                     -> ObjectSystem.LinkID
                        -> ObjectSystem.EventInstanceID list
val get_evtins : DynamicModel.EventID
                 -> ObjectSystem.EventInstanceID list
                    -> ObjectSystem.EventInstanceID
val ifPositive : int -> bool
val internalobjects : InternalObjects ref
val lookup_a1 : ObjectSystem.ObjectID -> int
val lookup_internalobjects : ObjectSystem.ObjectID
                             -> ObjectSystem.ObjectID Set
val lookup_linkOFobject : (''a * (''b * ''b)) list -> ''b -> ''a Set
val lookup_state : ObjectSystem.ObjectID -> DynamicModel.StateID
val message : string -> OS.Process.status
val new : ObjectSystem.ObjectID -> unit
val nop : unit -> unit
val notifyAttributeChanged : string * string * string * string
```

```
                                -> OS.Process.status
    val notifyStateChanged : string * string * string -> OS.Process.status
    val objects : ObjectSystem.ObjectID Set ref
    val removeEventOfLink : {event:ObjectSystem.EventInstanceID,
                             link:ObjectSystem.LinkID} list
                            -> ObjectSystem.LinkID
                               -> {event:ObjectSystem.EventInstanceID,
                                   link:ObjectSystem.LinkID} list
    val self : ObjectSystem.ObjectID ref
    val sendAttributeMessage_a1 : ObjectSystem.ObjectID -> OS.Process.status
    val sendStateMessage : ObjectSystem.ObjectID -> OS.Process.status
    val set_a1 : string -> int -> unit
    val set_state : string -> string -> unit
    val transition : ObjectSystem.ObjectID
                     -> ObjectSystem.EventInstanceID Set -> unit
    val warning : string -> unit
  end
```

ここで

```
val new : ObjectSystem.ObjectID -> unit
```

という関数が定義されている。ここで new 関数は実体化したオブジェクトを unit 型へ返すことが推測される。

クラス図において unit 型の属性を作成、定義し、この unit 型の属性を用いて新しいオブジェクトとリンクを発生させる。(図 3.59),(図 3.60)

図 3.59: ML の unit 型の属性の定義 1

次にオブジェクトを発生させたいイベントのアクションに以下の ML プログラムを追加する。(図 3.61)、(図 3.62)

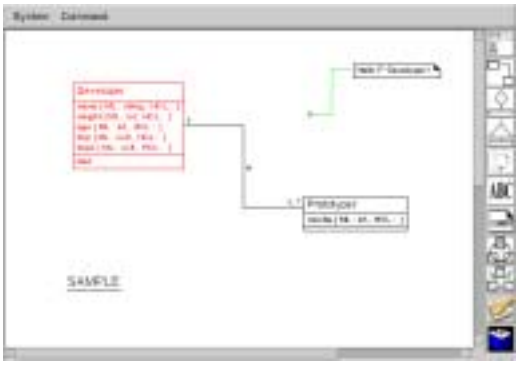

図 3.60: ML の unit 型の属性の定義 2

図 3.61: ML の unit 型の属性の定義 3

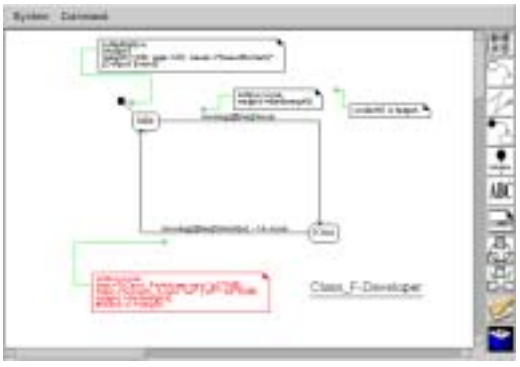

図 3.62: ML の unit 型の属性の定義 4

```
tmp:=%Class_Prototyper.new "p2"%$$;
tmp2:=%Assoc_A.new "a2" ("d1","p2")%&&
```

このプログラムは属性 tmp に「新しいオブジェクト p2 を実体化させている。属性 tmp2 に新しいリンク a2 を実体化させている。」とう意味である。
オブジェクトの生成の記法は、**Class_クラス名.new "オブジェクト名"**である。リンクの生成の記法は、**Assoc_関連名.new "リンク名" ("オブジェクト a","オブジェクト b")** であり、オブジェクト a とオブジェクト b 間にリンクインスタンスを生成するという意味である。

ここで%%で挟まれている部分には ML 言語を記述し、$$で囲まれている部分には HOL を記述するが、今回 HOL は用いないため何も書かない (図 3.61)

structure 構文の順序　例題モデルのアクション式に new 関数の表記を記述した後でプロトタイプ実行を行うとエラーが検出された。

エラー部分には

```
unbound structure: Class_Prototyper in path Class_Prototyper.new
```

と出力されている。図 3.63

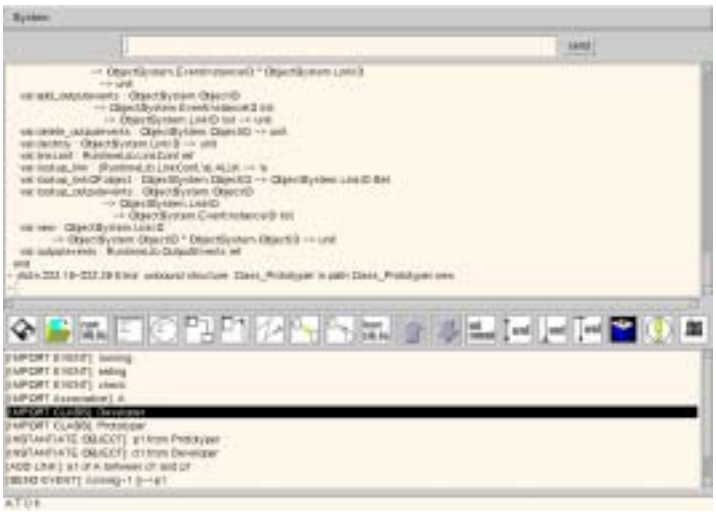

図 3.63: エラー表示 2

クラス Developer で Prototyper のオブジェクトを生成する記述をしたため、クラス Developer の structure よりも先に Prototyper の structure をインタプリタに読み込ませる必要がある。

実行系列の [IMPORT CLASS] コマンドの順序を入れ替え、再度インタプリタに読み込ませるとエラーもなく形成された。(図 3.64)

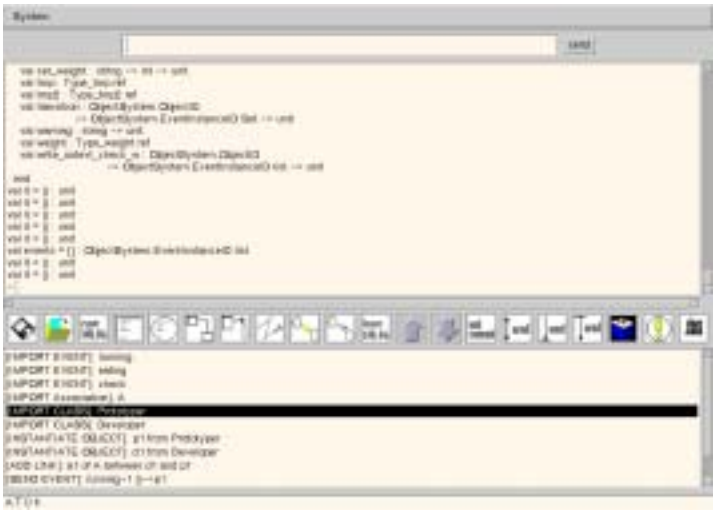

図 3.64: new 関数を交えた structure 構文の完成

new 関数で生成したオブジェクトやリンクはアニメーションとして表示されないため、オブジェクトが生成されているかどうか、SML システムによるプログラム対話で確認しなければならない。
前回、構築した実行系列 (図 3.51) を用いてイベントの送信と同期までプロトタイプ実行を行った。

```
[SEND EVENT]: running()->p1
```

を 2 回、

```
[SYNCHRONAIZATION]:from d1 to p1 using a1
```

を 1 回送信した。
次に、アクション式に記述した新しいオブジェクトが生成しているか確認を行うため、インタプリタ上に直接

```
Class_Prototyper.monita;
```

を入力してみる。結果、

```
- Class_Prototyper.monita;
val it = ref[("p1,98"),("p2,0")]:Class_Prototyper.Type_monita ref
```

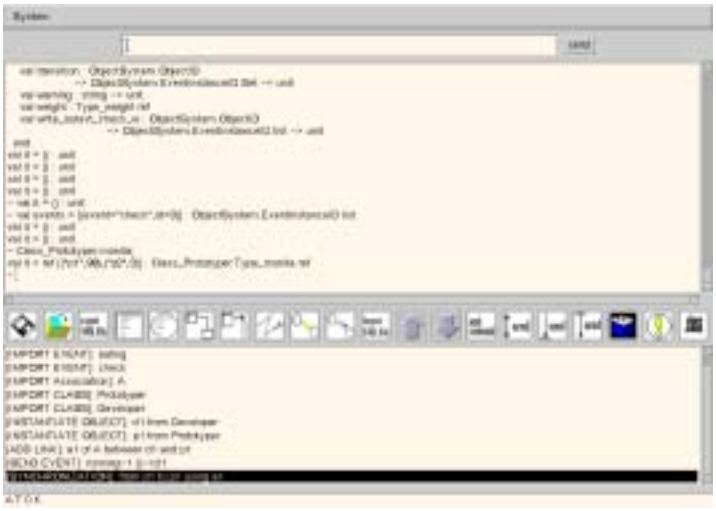

図 3.65: 生成させたオブジェクトの確認

と出力され p2 オブジェクトの生成が確認できた。(図 3.65)
　また直接インタプリタ上で ML プログラムを使用してオブジェクトを生成させるとも可能である。

```
Class_Prototyper.new("p3") ;
```

(図 3.66)

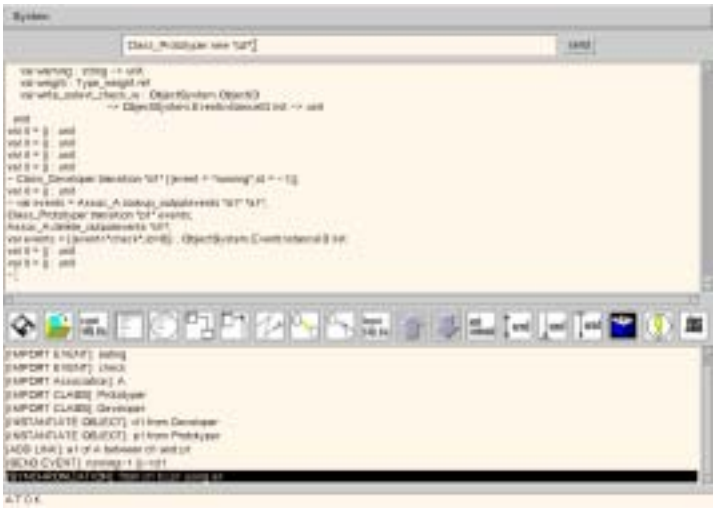

図 3.66: 開発者によるオブジェクト生成 2

　対話型プログラムで確認すると p3 オブジェクトの生成が確認された。(図 3.67)

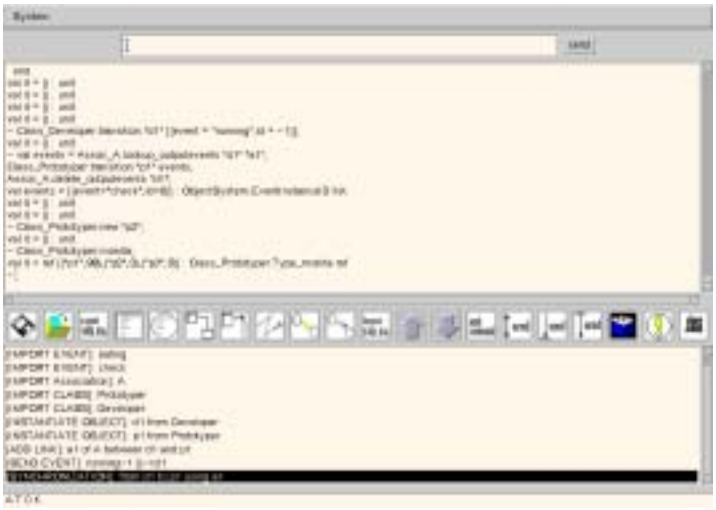

図 3.67: 生成させたオブジェクトの確認 2

第4章　携帯情報端末の開発実験

4.1　仕様

開発対象である携帯情報端末システムの論理的構造と振る舞いを明らかにし、分析モデルを作成する。携帯情報端末の操作部インターフェースと、携帯情報端末ネットワークを含めたシステムをモデル化する。

操作部インタフェース　携帯情報端末には、入力部としてのボタンと、出力部としてのモニタ、スピーカーがある。

ダイヤルボタンを押すとモニタに押された数字番号が表示される。ここで、数字ボタンを押すという操作を入力イベントとし、モニタに数字を表示させることをアクションとすることで、状態遷移を表現することができる。(図 4.1)

携帯電話ネットワーク　携帯情報端末の基本機能として、電話通信がある。

電話通信は携帯電話ネットワークを経由して行われる。(図 4.2)

携帯電話に関する情報は加入者情報データベースで間接的に管理されている。携帯電話の通信を直接に受けるのは、無線アクセスネットワーク内にある基地局である。ここで携帯電話は基地局とに 1..*:1 に対応している。携帯電話が現在どの基地局上にいるのか絶えず電波を飛ばして位置情報を確認している。

(図 4.3) は以上の対応関係を表している。

メール機能による通信も同様に、基地局を経由するが、ベースステーションコントローラによって通信の進路がかわり、IP ネットワークに接続される。(図 4.2)

以上の携帯電話ネットワークは開発途中である。臨時としてデータベースの機能をベースステーションに持たせてあるため、モデルは現実社会の本来の振る舞いを表現していない。

4.2　モデル

携帯情報端末を対象にしてモデルのプロトタイプ実行実験を行った。(図 4.4),(図 4.5),(図 4.6),(図 4.7),(図 4.8),(図 4.9),(図 4.10),(図 4.11),(図 4.12),(図 4.13),(図 4.14),(図 4.15)

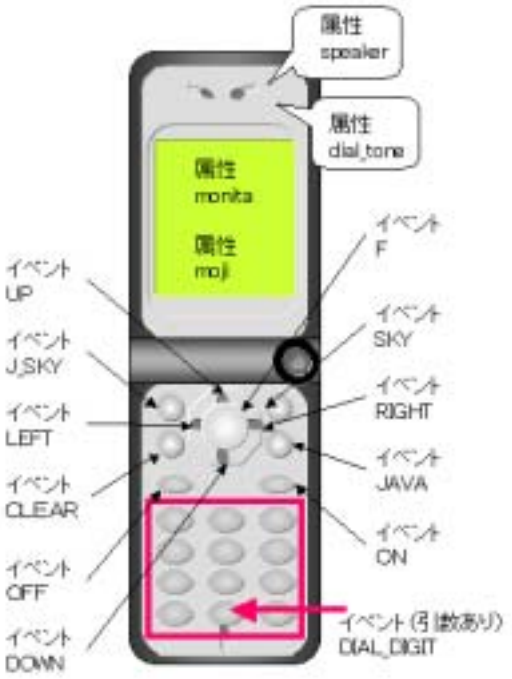

図 4.1: 操作部インタフェースの仕様

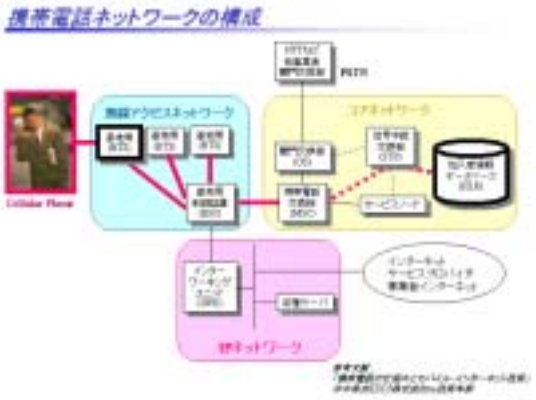

図 4.2: 携帯電話ネットワークの構成

図 4.3: 携帯電話ネットワークの構成 2

クラス図 4.4 の中心的な役割を担っているのは *MobilePhone* クラスで、アニメーションで振る舞いを確認しやすいよう、1 つのクラスに状態遷移図を意図的に集中させモデル構築を行った。モデルの再利用という点は配慮していない。
現システムは、ダイヤルプッシュから始まる基本的な電話発着機能、リダイヤル機能、アシスト機能、音量設定機能、各種メニュー表示とその説明機能、簡単なメール送信機能を持つ。
現システムは未完成であり、従来の携帯情報端末の振る舞いにはほど遠い。例えば、クラス *BaseStation* は電話番号の検索機能を持たせてあるが、本来のベースステーションにはこのような機能はない。また使われていないクラスもある。現システムで機能しているのはクラスは、クラス *MobilePhone*、クラス *BaseStation*、クラス *Mail*、クラス *PersonID*、クラス *AssistMenu*、の５つである。

クラス *MobilePhone* とクラス *BaseStation* 間のリンク *A* は電話の通信をやりとりするものである。クラス *MobilePhone* にリンクしているクラス *Mail*、クラス *PersonID*、クラス *AssistMenu* はそれぞれ自身のクラスのインスタンスを生成する。
ここで、クラス *MobilePhone* のインスタンス *mp1* から、同じくクラス *MobilePhone* のインスタンス *mp2* へダイヤルプッシュ形式で電話をかける電話発着機能について説明する。

ダイヤルプッシュにて電話をかける電話発着機能　図 4.5

- クラス *MobilePhone* の状態遷移
  - 初期設定
    MPID:=[];
    dial_tone:=”silence”;
    numberList:=[];

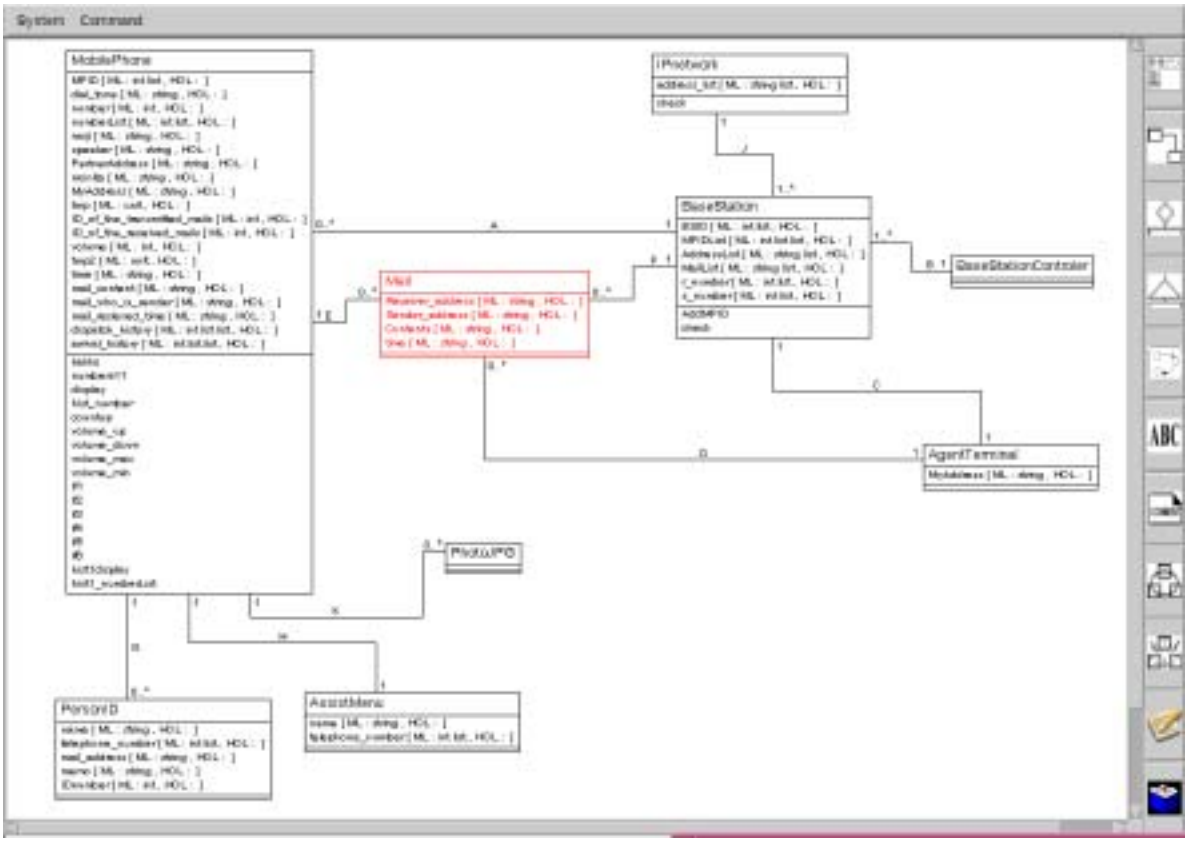

図 4.4: 携帯情報端末モデルのクラス図

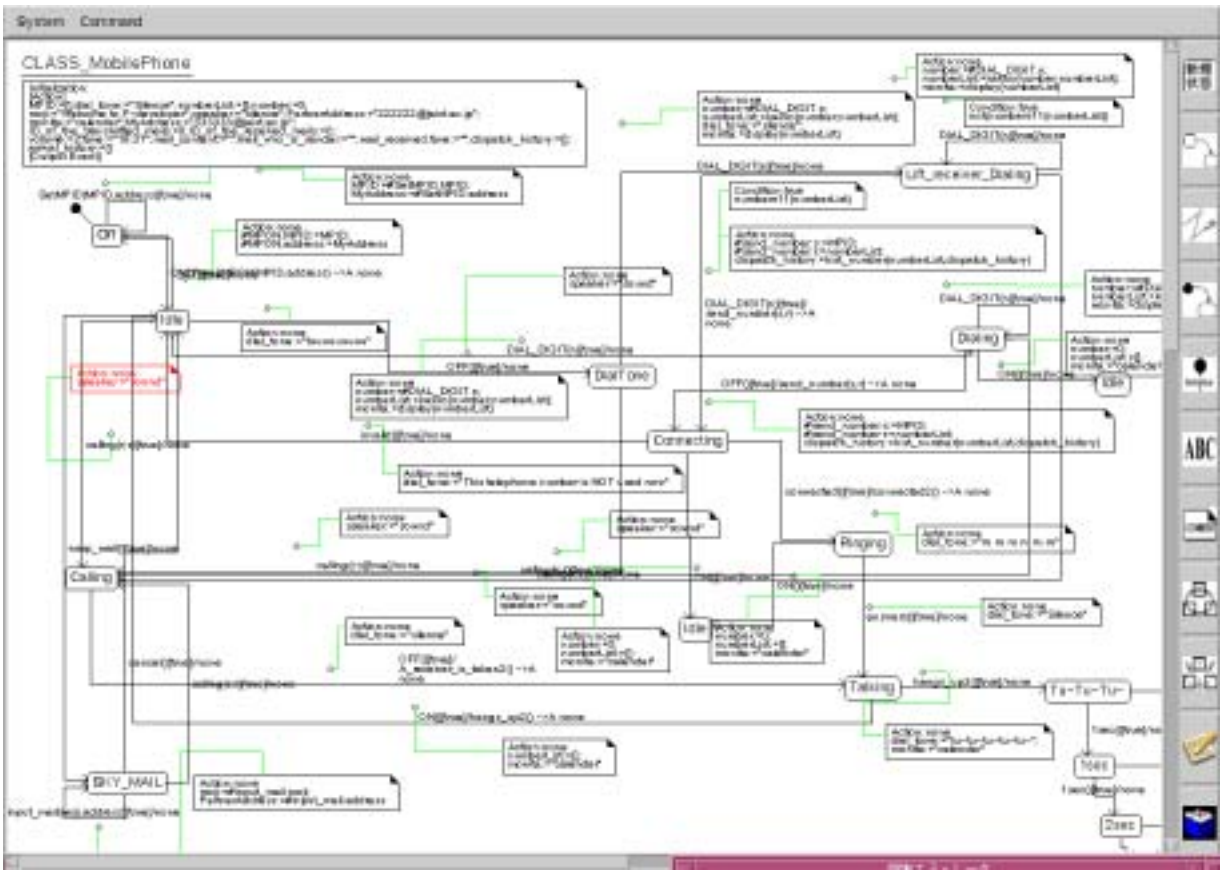

図 4.5: *MobilePhone* クラスの状態遷移図 1

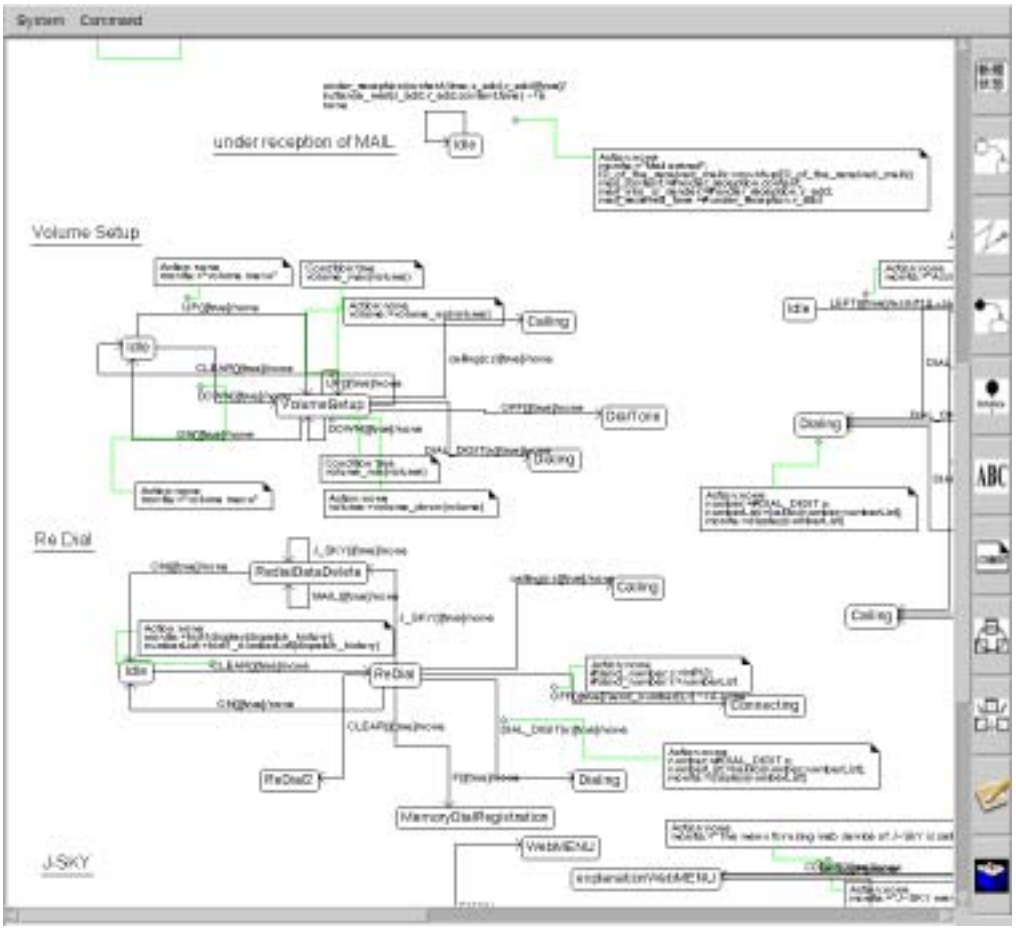

図 4.6: *MobilePhone* クラスの状態遷移図 2

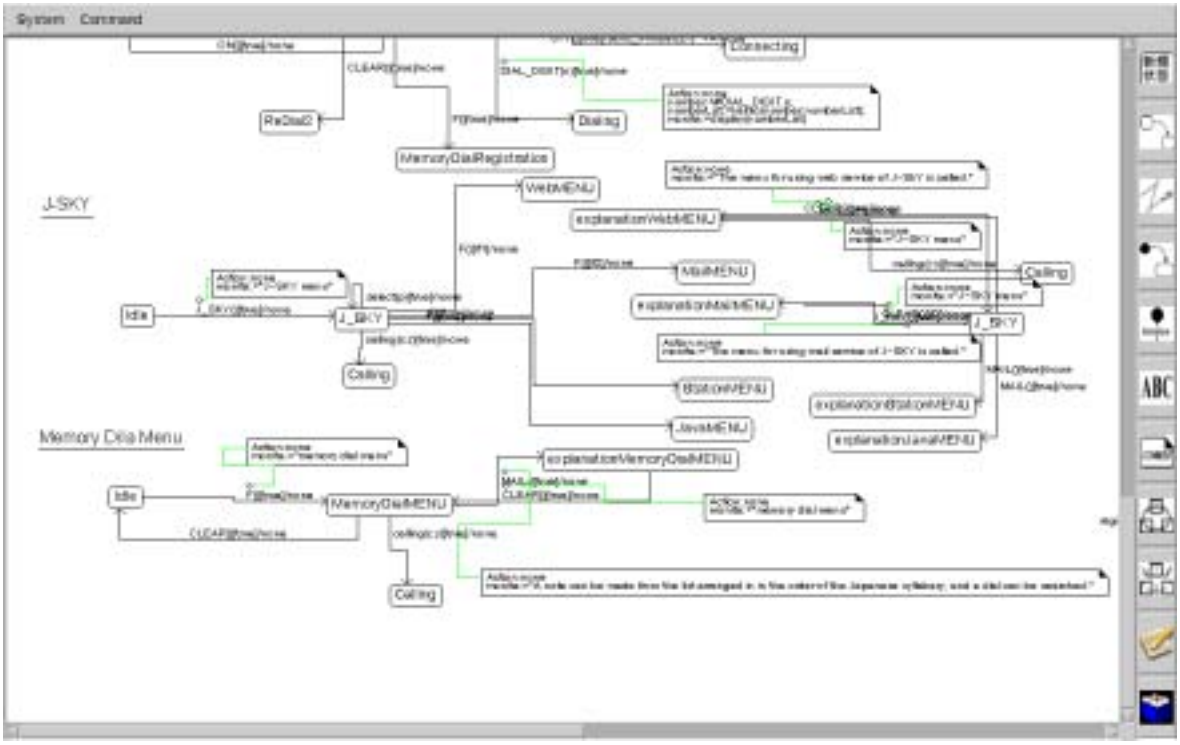

図 4.7: *MobilePhone* クラスの状態遷移図 3

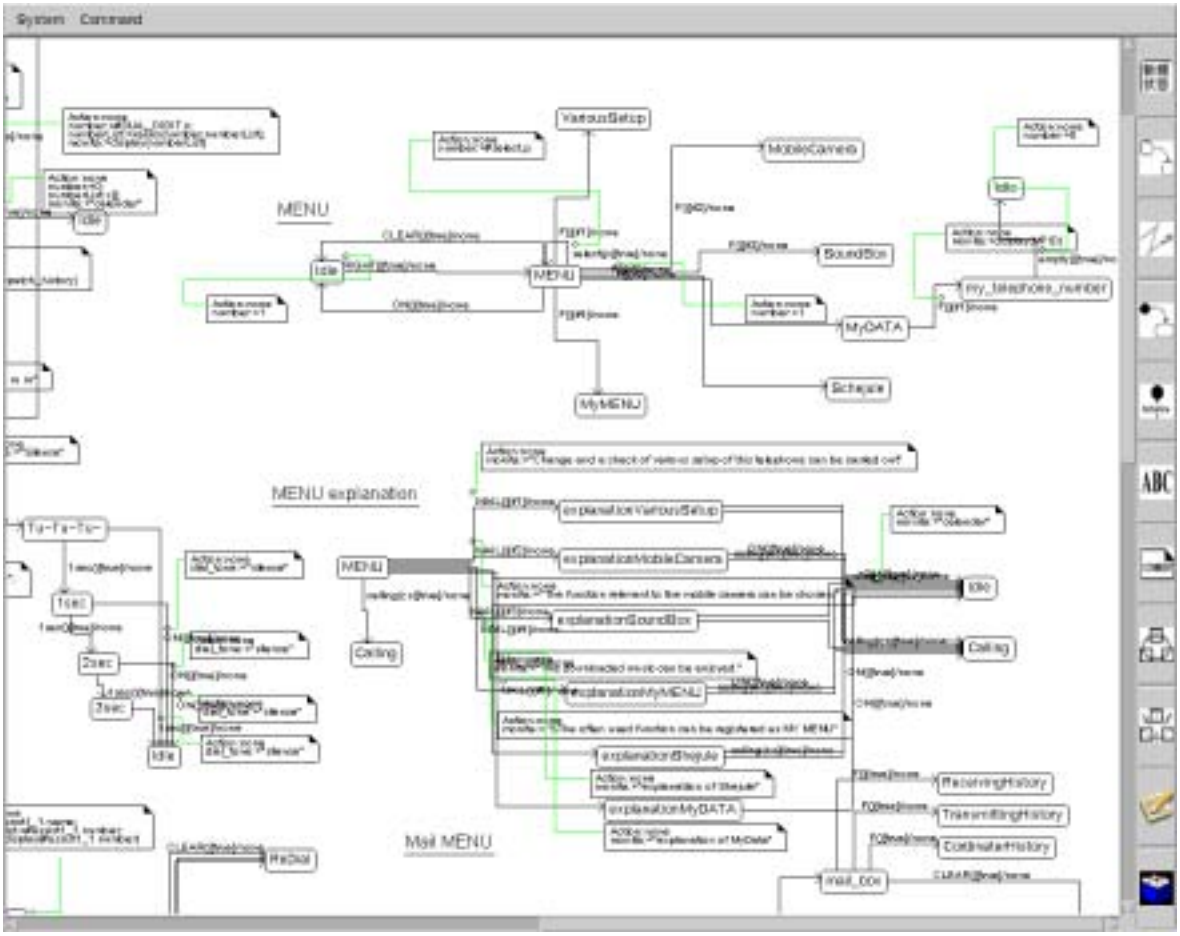

図 4.8: *MobilePhone* クラスの状態遷移図 4

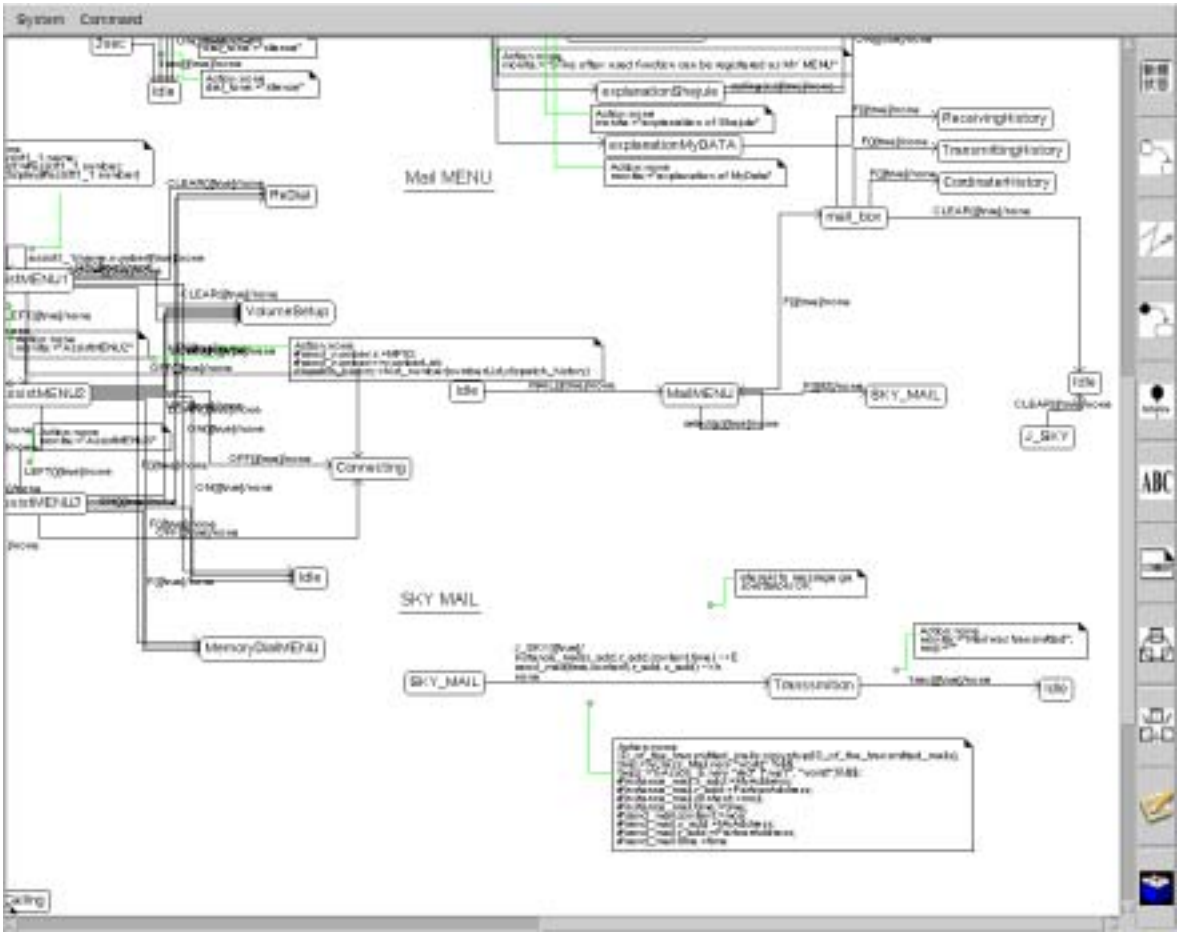

図 4.9: *MobilePhone* クラスの状態遷移図 5

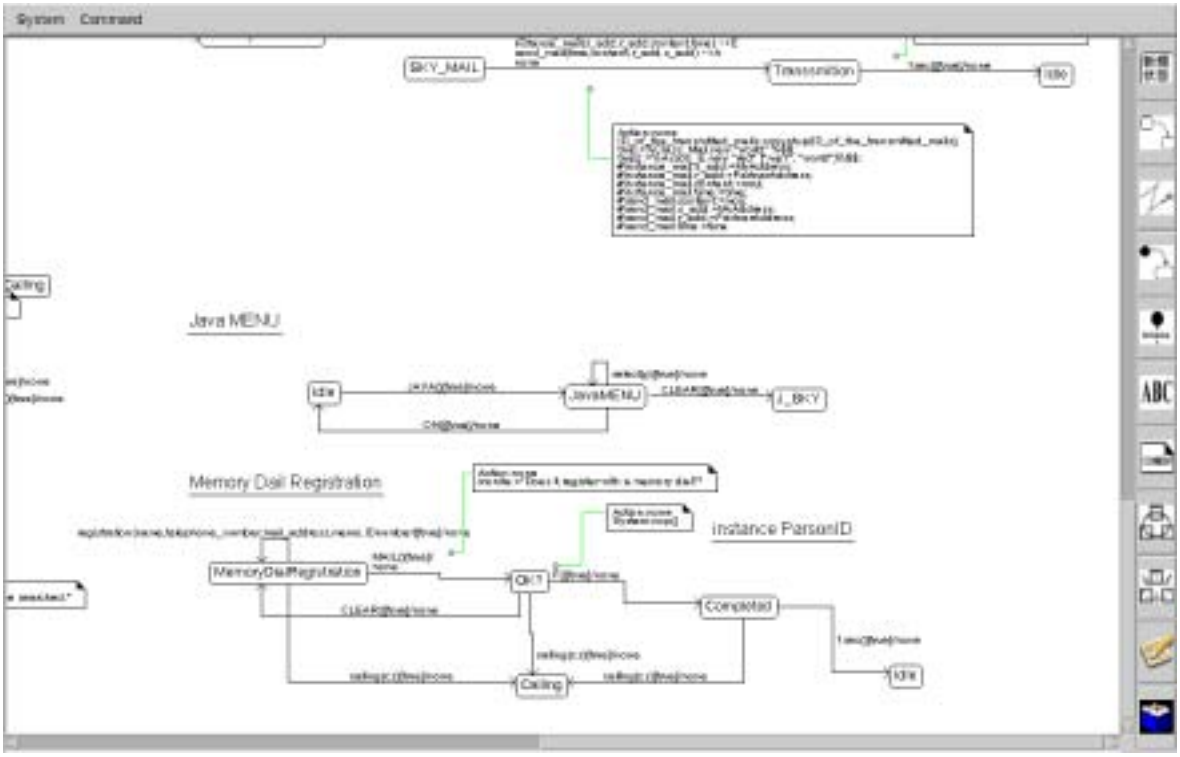

図 4.10: *MobilePhone* クラスの状態遷移図 6

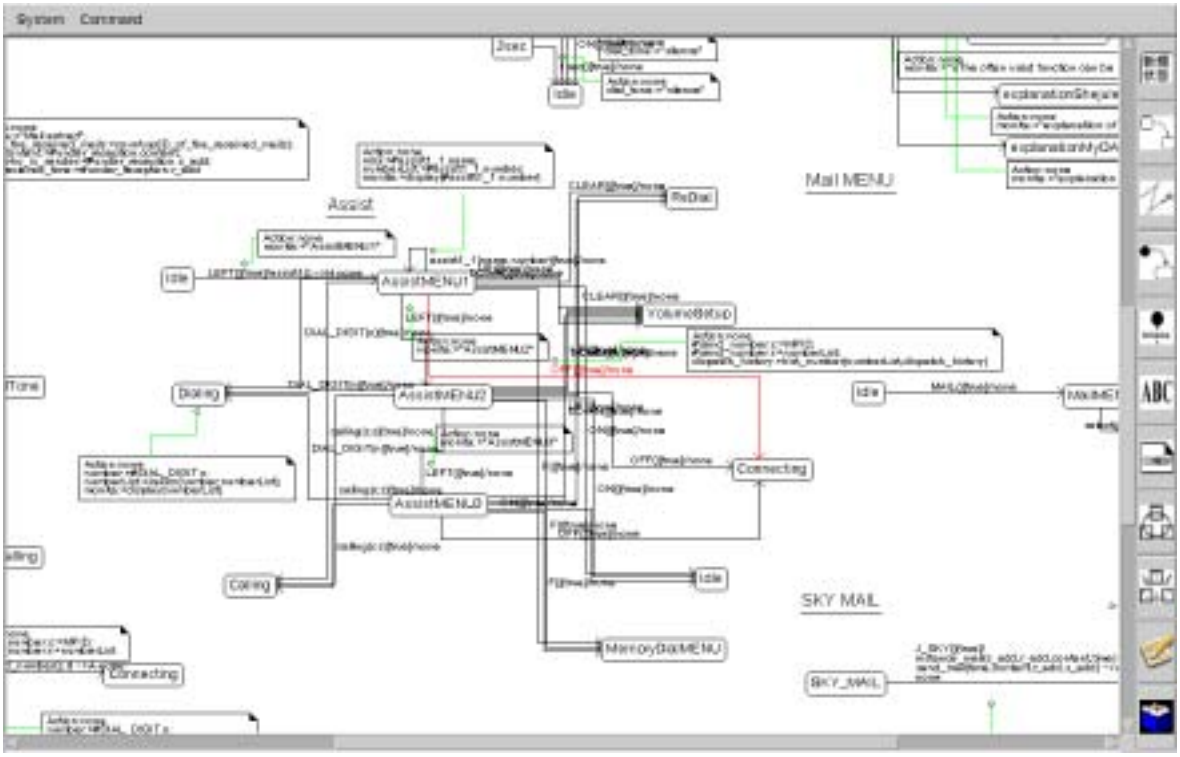

図 4.11: *MobilePhone* クラスの状態遷移図 7

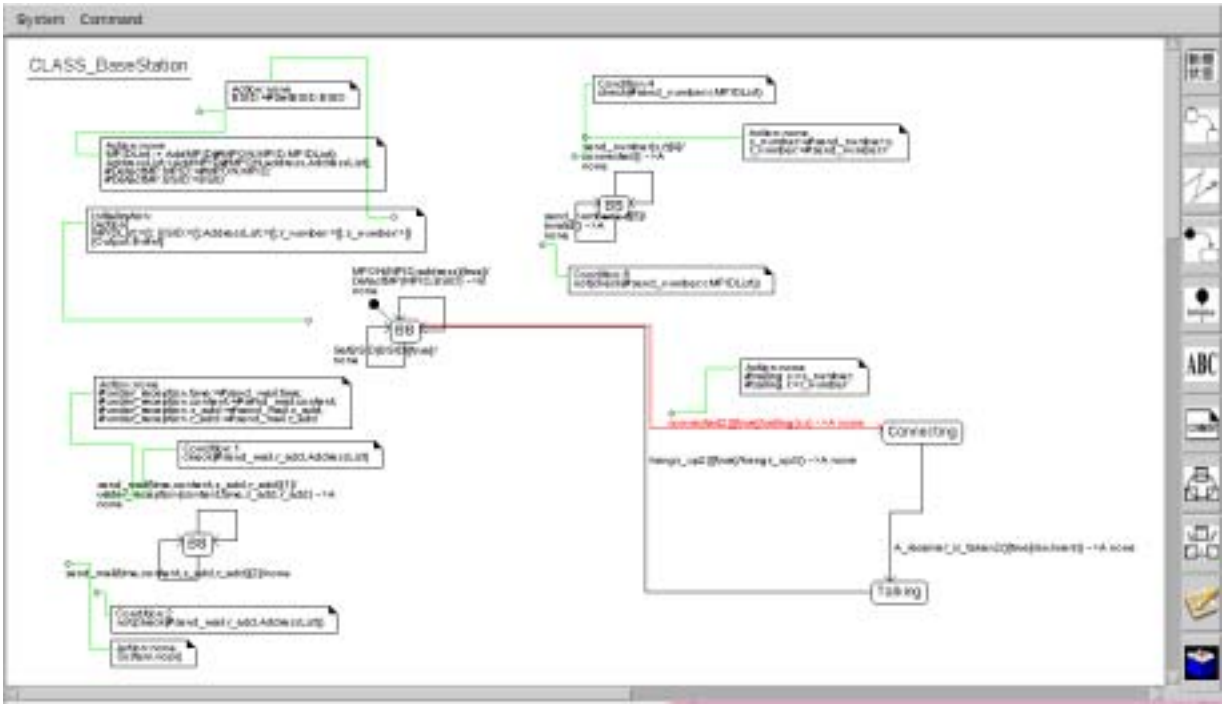

図 4.12: *BaseStation* クラスの状態遷移図

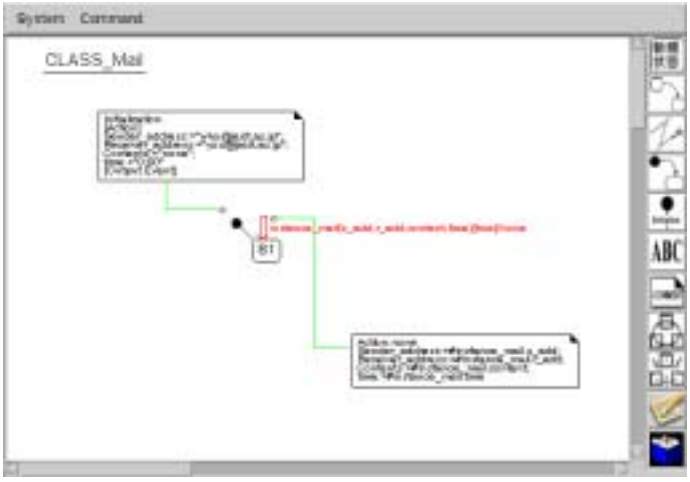

図 4.13: *Mail* クラスの状態遷移図

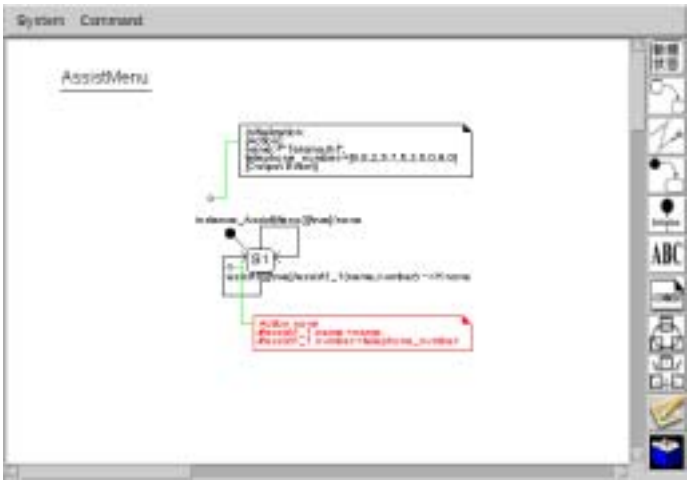

図 4.14: *AssistMenu* クラスの状態遷移図

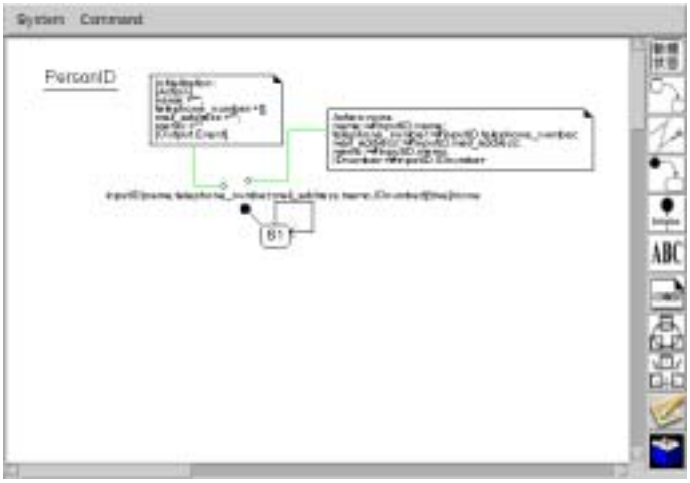

図 4.15: *personID* クラスの状態遷移図

表 4.1: クラス *MobilePhone* の属性

属性名	ML 型	説明
MPID	int list	自分の ID である電話番号
dial_tone	string	通常のインタフェースで確認可能な [ダイヤルトーン]
number	int	0-9 までの 1 桁の整数
numberList	int list	相手先の電話番号のバッフ
moji	string	通常のインタフェースで確認可能な [文字]
speaker	string	通常のインタフェースで確認可能な [スピーカー]
PartnerAddress	string	相手先の電子メールアドレス
monita	string	通常のインタフェースで確認可能な [モニター]
MyAddress	string	自分の端末の電子メールアドレス
tmp	unit	ML で直接オブジェクトを生成するために必要
tmp2	unit	ML で直接リンクを生成するために必要
ID_of_the_trasmitted_mail	int	メール送信履歴の ID 番号
ID_of_the_received_mail	int	メール受信履歴の ID 番号
volume	int	1-6 までの 1 桁の正数。受話音量レベル
time	string	通常のインタフェースで確認可能な [時間]
mail_content	string	メールの中身
mail_who_is_sender	string	メールの送信者の名前、または電子メールアドレス
mail_received_time	string	相手がメールを送信した時間
dispatch_history	int list list	電話の発信履歴のバッファ
arrival_history	int list list	電話の着信履歴のバッファ

表 4.2: クラス *MobilePhone* の関数

関数名	ML 関数
kakko	fun kakko(i,k)=i::k
numbers11	fun numbers11 m = if (List.length(m)) = 11 then true else false
display	fun display (a)= String.concat(map Int.toString(List.rev(a)))
hist_number	local fun saigo m = List.take(m, length(m)-1) in fun hist_number (a, m) = if length(m)¡5 then a::m else a::(saigo(m)) end
countup	fun countup(i) = i+1
volume_up	fun volume_up (i)=i+1
volume_down	fun volume_down (i) = i-1
volume_max	fun volume_max (i) = if i<6 then true else false
volume_min	fun volume_min (i)= if i>1 then true else false
if1	fun if1 (i) = if i = 1 then true else false
if2	fun if1 (i) = if i = 2 then true else false
if3	fun if1 (i) = if i = 3 then true else false
if4	fun if1 (i) = if i = 4 then true else false
if5	fun if1 (i) = if i = 5 then true else false
if6	fun if1 (i) = if i = 6 then true else false
hist1display	fun hist1display (m) = String.concat(map Int.toString(List.rev(List.nth((m),0))))
hist1_numberList	fun hist1_numberList (m)= List.nth((m),0)

表 4.3: クラス *BaseStation* の属性

属性名	ML 型	説明
BSID	int list	自分の ID。*BaseStationContoler* クラス（未設定）で管理される
MPIDList	int list list	クラス *MobilePhone* の ID を管理するバッファ
AddressList	string list	クラス *MobilePhone* のメールアドレスを管理するバッファ
MailList	string list	クラス *Mail* 関連アドレス管理。(作成途中)
r_number	int list	電話発信側の電話番号
s_number	int list	電話着信側の電話番号

表 4.4: クラス *BaseStation* の関数

関数名	ML 関数	説明
AddMPID	fun AddMPID(m,l)=m::l	バッファにデータを格納する
check	fun check(m,l)=List.exists(fn y => x=y) l	バッファのデータを確認する

表 4.5: クラス *Mail* の属性

属性名	ML 型	説明
Receiver_address	string	受信者のメールアドレス
Sender_address	string	送信者のメールアドレス
Contents	string	メールの中身
time	string	メールを送受信した時間

number:=0;
moji:="Welcome to F-Developer";
speaker:="silence";
PartnerAddress:="222222@jaist.ac.jp";
monita:="calender";
MyAddress:="333333@jaist.ac.jp";
ID_of_the_transmitted_mails:=0;
ID_of_the_received_mails:=0;
volume:=3;
time:="18:37";
mail_content:="";
mail_who_is_sender:="";
mail_received_time:="";
dispatch_histry:=[];
arrival_history:=[]

– 状態 Off から Off への遷移
ここでの状態遷移は携帯情報端末の出荷時に端末自身の ID（ここでは携帯番号と電子メールアドレス）を記録していることを意味する。

  * 入力イベント
    SetMPID: 引数 MPID、address を持つ。
  * アクション

属性 MPID にイベント SetMPID の引数 MPID を設定する。
属性 MyAddress にイベント SetMPID の引数 address を設定する。

– 状態 Off から Idle への遷移
ここでの状態遷移は携帯電話の電源を入れた際に一番近くのベースステーションに自身の存在（ID）を知らせていることを意味する。このイベントによりこれを関知したベースステーションはデータベースにその情報を蓄積する。（クラス *BaseStation* と同期している）

* 入力イベント
ON
* アクション
イベント MPON の引数 MPID に属性 MPID を設定する。
イベント MPON の引数 address に属性 MyAddress を設定する。
* 出力イベント
MPON: 引数 MPID、address を持つ。関連 A にそって送出。

– 状態 Idle から Dialing への遷移
ここでの状態遷移は携帯電話のダイヤルをプッシュしはじめたことを意味する。本来のシステム開発では Idle 状態から 12 個の数文字

```
1,2,3,4,5,6,7,8,9,0,*,#
```

を割り当てた 12 個の状態に分岐しなければならないが、モデルが複雑になってしまうため今回は取りやめる。イベントの引数に

```
1,2,3,4,5,6,7,8,9,0
```

のどれか１つ整数を書くこととする。
これから携帯電話からプッシュされ続けてる整数をバッファに貯めておき、モニタにはその番号を表示させる。

* 入力イベント
DIAL_DIGIT: 引数 n を持つ。
* アクション
属性 number にイベント DIAL_DIGIT の引数 n を設定する。
関数 kakko により、属性 numberList[int list] に属性 number[int] の値をためる。
関数 display により、属性 numberList の値を属性 monita に設定する。
* 出力イベント
なし。

– 状態 Dialing から Dialing への遷移
同上。携帯電話からプッシュされ続けている整数をバッファに貯めておき、モ

ニタにはその番号を表示させる。プッシュを途中でやめてIdle状態にすることもできる。本来のシステムではここで一定時間放置しておくとIdle状態に戻る時間制約がある。

* 入力イベント
  DIAL_DIGIT: 引数nを持つ。
* アクション
  属性numberにイベントDIAL_DIGITの引数nを設定する。
  関数kakkoにより、属性numberList[int list]に属性number[int]の値をためる。
  関数displayにより、属性numberListの値を属性monitaに設定する。
* 出力イベント
  なし。

– 状態DialingからConnectingへの遷移
ここでは送信ボタンを押したことを意味する。ダイヤルトーンが聞こえ、発信履歴に番号を残す。同期して、自分のID（携帯番号）(mp1.MPID)、相手先の電話番号（mp2.MPID）を同時にクラス*BaseStation*に伝える。本来のシステムはここで発信時間も伝わる。
同期してBaseStationは自分のデータベースに相手先の電話番号（mp2.MPID）がないか探す。相手先の電話番号が確認されたとき、そのBaseStationにぶら下がっている携帯情報端末全てに、かけている側の電話番号（mp1,MPID）と相手先の電話番号（mp2.MPID）を発信する。
さらに同期して相手先の電話番号(mp2.MPID)が自分のID(mp2.MPID)と一致した場合、mp2の携帯情報端末のモニタに書けている側の電話番号(mp1.MPID)を表示し、スピーカーに着信メロディを鳴らす。
本来のベースステーションはただのアンテナで電話番号などの情報は格納してはいない。情報を管理しているデータベースは他にある
相手先の電話番号が見つかった場合、発信側(mp1.MPID)にも同期してダイヤルトーンが「*ルルルル。ルルルル。*」と聞こえるようになる。

* 入力イベント
  OFF
* アクション
  イベントsend_numberの引数sに属性MPIDを設定する。
  イベントsend_numberの引数rに属性numberListを設定する。
  v関数hist_numberにより、属性dispatch_history[int list list]に属性numberList[int list]の値をためる。
* 出力イベント
  send_number: 引数にs、rを持つ。関連Aにそって送出。

– 状態 Connecting から Ringing への遷移
この遷移はベースステーションから同期メッセージとして伝わる。相手先の電話番号が見つかった場合、発信側 (mp1) にも同期してダイヤルトーンが「ルルルル。ルルルル。」と聞こえるようになる。
  * 入力イベント
    connected
  * アクション
    属性 dial_tone に”ru ru ru ru ru ru” を設定。
  * 出力イベント
    connected2: 関連 A にそって送出。
– 状態 Ringing から Talking への遷移
電話先の相手が受話器をとった場合、相手先の着信メロディは消える。この時、同期が発生し、ベースステーションを経由して、A 側も通話が可能な状態になる。このときダイヤルトーンは消える。この時、かけている側 (mp1) も相手先 (mp2) も同じ状態 Taking いる。
  * 入力イベント
    answers
  * アクション
    属性 dial_tone に”silence” を設定。
  * 出力イベント
    なし。
– 状態 Talking から Idle への遷移
通話を終え、先に受話器をおいた方は再び Idle 状態に戻る。通話中やダイヤルプッシュ時も途中で受話器をおくと Idle 状態に戻る。
取り残された側はダイヤルトーン「ツーツーツー」が聞こえるようになる。ここでも数秒立ったら Idle 状態に戻るという状態遷移図を書きたかったが、状態遷移図では時間を表すことが困難であったため状態 Tu-Tu-Tu-、状態 1sec と表現するしかなかった。
またクラス *BaseStation* は携帯電話が通話中に電話がかかってこないよう状態遷移を作った。しかしこの状態遷移図では一組の携帯電話しか通話する状況になれない。作り直すことが今後の課題である。
  * 入力イベント
    ON
  * アクション
    v 属性 monita に”calender” を設定。属性 numberList を [] 設定。
  * 出力イベント
    hangs_up2

携帯情報端末モデルは未完成であるが、現時点のモデルでプロトタイプ実行は予測した通りの状態遷移を確認している。図 4.16 図 4.17
new 関数を用いてのクラス *Mail* オブジェクトの生成も確認できている。

システム上のエラーも確認されず、F-Prototyper は既にでも開発現場で実用可能であることが確認された。

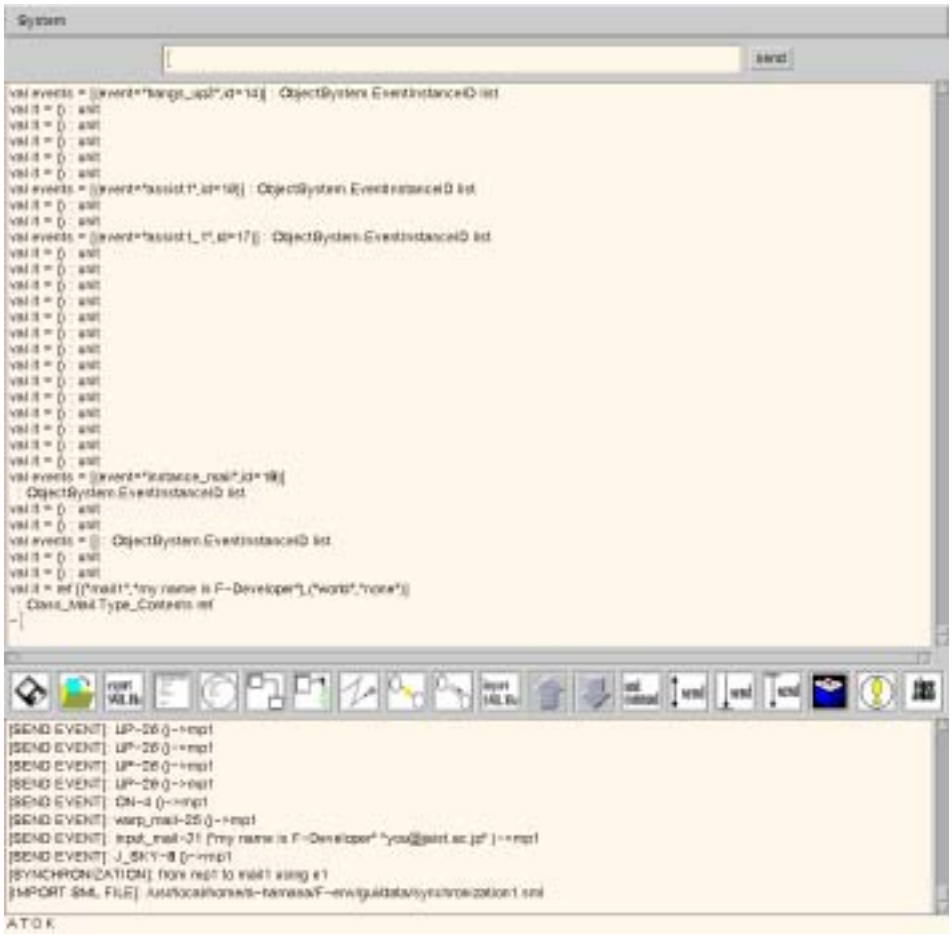

図 4.16: 携帯情報端末モデルのプロトタイプ実行の様子

ボリュームの設定　携帯情報端末上で十字キーでボリュームの設定を行う。(図 4.6)

初期状態で設定されている通話の音量ボリュームのレベルは３とし、レベル１からレベル６までの範囲で設定を行う。

入力イベントは UP、もしくは DOWN であり、アクションは属性値 volume の変化を行う。ここでガード条件 volume_max,volume_min を用いる。(表 4.2)

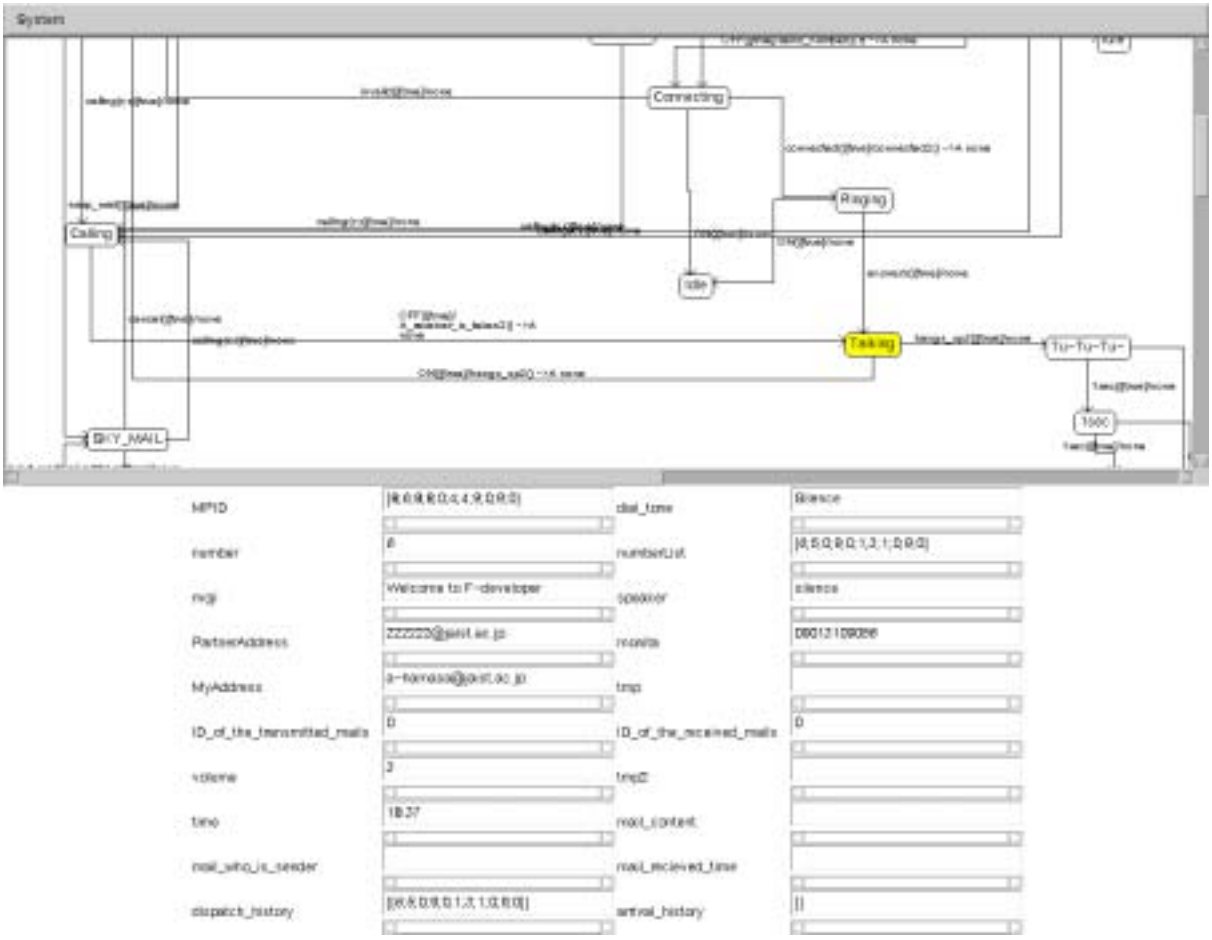

図 4.17: 携帯情報端末モデルの振る舞いアニメーション

リダイヤル　一度発信した電話番号を格納しておく、新しい番号順に電話番号を引き出し、再度発信することができる。(図 4.6)

メールの受信　送られてきたメールアドレスが自分のアドレスと一致した場合、*MobilePhone* クラスの属性値 mail_tontent,mail_who_is_sender,mail_received_time にそれぞれ送られてきた情報を表示する。この時、同期としてメールクラスのインスタンスを作成し、送られてきた属性値を格納する。現在作成中である。(図 4.6)

J-SKY　ある企業の携帯用コンテンツである。ボタン１つで呼び出すことが可能である。(図 4.7)

メモリダイヤルメニュー　生成したメモリダイヤル *PersonID* インスタンスを呼び込む。現在制作中である。(図 4.7)

メニュー画面　メニュー画面は画面上に複数のアイコンを並べてある。アイコンを選択するには左右の十字キーを数回動かし、F キーを押すと遷移する。(図 4.8)

メニュー画面の説明表示　メニュー画面サブメニュー画面を選択する場合、アイコンの意味を表示させたい。アイコンを選択しボタンを押すと画面上に説明文が表示される。(図 4.8)

スカイメール　メール送信機能。作成途中である。(図 4.9)

Java メニュー　Java 言語を用いたゲーム等のメニュー画面。(図 4.10)

メモリダイヤルの記録　ここで *PersonalID* オブジェクトを生成させ、個人情報を記録する。(図 4.10)

アシストメニューの表示　ボタン数回で記録している番号へ電話をかける機能。(図 4.11)

4.3 考察

4.3.1 操作法習得の障壁

今回の実験は、モデル開発者にとってF-Develoerの操作法を習得する際、どこに障壁があるかを調査することも兼ねて行った。以下、F-Prototyperを習得する際の経験をまとめる。

操作法習得の初期段階

- 操作法習得の障壁は高い。マニュアル、及び、実装者の助けがなければ全く対応できない。
- 理論的な背景を知っておくことが必要である。
- 各種アイコンには説明表示が欲しい。
- マニュアルがあったとしてもF-Modelの記述を習得することができない。
- プロトタイプ実行の意味を直感的に理解できない。
- モデルエディタの操作法は直感的で容易である。
- F-Prototyperの操作法は全く理解できない。したがって、アイコンも直感的に理解しにくい。
- 中期段階に至るまでに非常に時間がかかる。

操作法習得の中期段階

- マニュアルが読めるようになる。
- マニュアルに従い非常に小さなF-Modelをプロトタイプ実行することができる。
- F-Prototyperの操作法を習得できる。
- F-Prototyperのバックグラウンドでどのような支援を行っているか直感的だが理解できるようになる。
- インタプリタ上のエラーをモデル上で発見するのに時間がかかる。
- インタプリタは実装者の助けを得なければ訂正個所を発見できない場合がある。
- インタプリタ上のエラーに比べ、アニメーションによる振る舞いの訂正は、直感的なため、比較的時間がかからない。
- 形式的手法理論について知識を必要としないことがわかる。
- プロトタイプ実行の有用性はまだ実感できていない。
- 終期段階に至るまでに時間がかかる。

操作法終期段階

- マニュアルなしでモデルを構築できる。
- プロトタイプ実行の高い有用性を理解できるようになる。
- モデルの記述上の誤りを即時に発見でき、巨大なモデルを構築できるまで時間がかからない。
- 振り返ってみるとインターフェースは使いやすい。
- ML による new 関数など高い技術を使用することできるようになる。
- インタプリタに直接 ML を記述できるようになる。

初期段階においてはマニュアルがなければ F-Developer の概要さえ掴めない。現在、F-Developer のマニュアルは存在していないため、本稿、第 3 章はマニュアルの意味も含めて記述した。中期段階においては、F-Model の記述に慣れるまでに多少の時間がかかってしまう。マニュアルにしたがって十分な例題をこなすことが望ましい。終期段階においては、次の節で述べている。

今後の課題として、プロトタイプ実行のクラスローダー機能の操作法を調査する。

4.3.2 携帯情報端末モデル開発実験を終えて

- 直接 ML のコードを書いてオブジェクトやリンクを生成/消滅も可能であったり、特定のオブジェクトから特定の属性値を引き出すことが可能であったり、従来の UML の限界を補っていた。
- (図 4.5),(図 4.6),(図 4.7),(図 4.8),(図 4.9),(図 4.10),(図 4.11), は同一の状態遷移図であるが、機能ごとに複数に分けて書いている。こられは独立な状態遷移ではなく、同じ名前の状態は同一のものである。

  ここで状態遷移図の再利用性が行えると考えられる。簡単な機能の追加を行いたい場合、状態遷移図のみを追加すればよい場合も考えられる。

  また機能毎に GUI を割当てられるとモデルの管理がしやすい。この機能にだけあらわれるイベント structure を一度に ML プログラムに読み込む機能が欲しい。
- サブ状態を記入できない。携帯情報端末モデルは作業中の全ての状態に Calling 状態を結び付けなければならない。これでは状態遷移が複雑になってしまう。作業中という枠組みを作り、１つの状態とし、Calling １つに遷移したい。また、サブ状態を再利用できる機能が欲しい。
- 今回は行えなかったが、モデルで作成したイベントをモデルの外部に送信することが可能である。図 4.1 は Microsoft 社のプレゼンテーション用ソフト Power

Point を使用して描いた。この図 4.1 を Web において、F-Developer 上からイベントを送信することが可能である。図 4.1 画面を通して実行を確認できるとすれば、その有用性はますます高くなる。今後、Power Point のようなドローツールと連携させることを提案する。

- 状態遷移図の規模が大きいこのようなシステムは、機能毎に状態遷移を確認できる実行系列を保存しておきたい。このための補助機能として coms ファイルの保存/読み込みは非常に有効である。ある状態の遷移を確認する場合、初期状態からイベントを動作させずにすむ。ここでいくつかの実行系列をまとめて保管するフォルダも自動生成する機能が欲しい。
- 自分で作成したモデルの振る舞いを確認できることは開発者のモチベーションを高める。そのためモデル構築の時間を短縮することが可能である。
- アニメーションによる状態遷移は直感的であることから教育効果を得ることも可能であると考える。
- 今回、クラスローダーという機能は使用しなかった。今後どのような機能であるのか調査を行い、マニュアルに追加したい。
- ツールの問題点として、モデルを描く速度が遅いのが気になる。
- 履歴のシステムが欲しい。例として、基地局から着信イベントである calling を受けとった場合、メールの作成中であっても状態 Calling へ遷移する。電話が終わった後にメールの続きを行いたい。このような履歴という
- 本来のシステムでは DIGIT$_D IAL$ ボタンは 12 個ある。この $DIGIT_D IAL$ というイベントをそのまま状態遷移図で表現しようとすると、その複雑度は大きく増す。メール機能による文字入力は本来この $DIGIT_D IAL$ を用いて行う。文字表記といった状態をどのようにモデル化するか考えている。

第5章　まとめ

F-Developer 習得には、理論的な知識が必要とされるため、最初のうちはモデル構築に時間を要した。また、インタプリタで発見される記述上のエラーが分析モデルのどの部分に対応するか発見するのに時間を必要とした。

モデルのプロトタイプ実行では状態遷移を視覚的に確認することができりため、開発者の大きな助けとなりうる。

結論としてオブジェクト指向分析段階において、F-Developer を使用とする有効性は高いが、ツール開発者の負担ができるだけ軽減するインターフェースを構築することが必要であると考える。

参考文献

[1] Toshiaki Aoki, Takuya Katayama: Prototype Execution of Independently Constructed Object-Oriented Analysis Model, Automationg the Object-Oriented Software Development Methods,ECOOP2001 Workshop,pp.25-33,2001.

[2] 青木利晃, 立石考彰, 片山卓也: 定理証明技術のオブジェクト指向分析への適用, pp.18-47, 日本ソフトウエア科学会学会誌 コンピュータソフトウエア Vol 18 No.4,July 2001

[3] 青木利晃, 片山卓也: オブジェクト指向分析モデルのプロトタイプ実行環境, オブジェクト指向シンポジウム'99, 情報処理学会,pp.161-168,1999.

謝辞

本研究の進行にあたり終始ご指導を頂いた片山卓也教授、青木利晃先生、矢竹健朗氏に深く感謝いたします。