拡張に適したアクティブソフトウェアの設計解析法

Analytical methods to design extensible Active software

渡邉勝正 † 井上晶広 † 蔵川 圭 † 中西正樹 † 山下 茂 †
Katsumasa WATANABE, Akihiro INOUE, Kei KURAKAWA,
Masaki NAKANISHI, Shigeru YAMASHITA

† 奈良先端科学技術大学院大学 情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology
{watanabe, akihi-i, kurakawa, m-naka, ger}@is.naist.jp

アクティブソフトウェアは自己を調整するという特徴をもつ．しかし，それを一般的に実現することは容易ではない．その可能性を高める方法のひとつとして，能動関数（active function）を用いてソフトウェアを構成する方法がある．本論文では，ソフトウェアが拡張されることを前提にして，拡張に適した設計とそのための解析法について述べる．状態遷移図と π 計算式を用いてソフトウェアの動きを表現し，それを基にして能動関数を用いてソフトウェアを構成する方法を提案する．設計対象の特徴による解析方法の適合性と，それに基づいて構成したソフトウェアの性能を，例に沿って論じる．性能を向上するためにはハードウェア回路の支援が有効であることと，提案する方法が可変部分を持つアーキテクチャ（reconfigulable architecture）の設計にも適用できることを論じる．

1 はじめに

ソフトウェアは，実行環境の変化や仕様の変更に対応して，変更や拡張がしやすいように設計と構成がなされていると，使用期間（life time）を長く保てる．それを実現する一つとして，アクティブソフトウェア（active software）[1] の考えがある．

アクティブソフトウェアは，自分の計算状況を知って，自分を調整する（aware and regulate）などの特徴をもつものである．そのためには,「いつ，どのような状態になったら，何をするか」を明確にしたソフトウェアの設計と構成の仕方が基本になる．その考えの実現として，われわれは能動関数（active function）[2] を基本にした設計法を提案している．

能動関数は，関数が自ら起動する条件（activation condition）を持ったもので，他から呼び出される（call）ことを待っているだけではない．そのために，ソフトウェアの構成を柔軟にして，変更や拡張を行ないやすくしている．しかし，反面,「どの能動関数が何時起動するか」を読み取り難くする弱点をもっている．それを緩和するために，能動関数をもつプログラム（.act）を C 言語のプログラム（.c）に変換して実行する過程に次の機能を準備した [2].

1. 静的な関数の参照関係：どの関数の中で，どの能動関数が起動する可能性をもつか．
2. 動的な関数の起動関係：実際に関数を呼び出している状態を実行中に追跡する．

本論文では，能動関数の起動に関連する部分の動きと関係を解析して形式的に記述する方法を考える．

起動条件を評価する指示と条件の成立を，一般に，事象の発生「! 送信」と事象の受理「? 受信」と観て，π 計算式で表現する．ただし，一つの送信に対して，それを受信するのは一つとは限らない．また，この関係を，拡張した状態遷移図で表して，直観的な理解を助ける．これに基づく解析と設計によって，変更や拡張を安全に安心して行なえることを計る．

ソフトウェアの拡張は，削除，追加，置換えの操作の集積である [3]．それを適切に行なうには，現在のソフトウェアの動きと構成を的確に把握することが肝要である．π 計算式と状態遷移図による表現はそれを支援する．

同時に，ソフトウェアの構造自体が拡張の操作に適していなければならない．能動関数による構成はそれに応える方法の一つである．

このような形式的記述から，能動関数をもったソフトウェアの枠組み（framework）を生成することができる．また，能動関数の起動を検出する回路を再構成可能なハードウェアで実現する記述に変換する可能性が得られる．

2 能動関数と起動文

能動関数（active function）は，自分が起動する条件（activation condition）を持つように，C 言語の関数の定義を拡張したものである．

定義 1：能動関数（aF）：

```
@ (起動条件_aC) ;
type 能動関数名_aF( [仮引数の並び] )
{ 関数本体の手続き }
```

□

計算の実行中に，起動条件（aC）が「真（**true**）」になると，関数の本体が起動する．

起動の仕方（起動モード）を起動文（activation statement）によって指定する．

定義 2：起動文（aS）：

```
起動モード　変数指定 ( [実引数の並び] ) ;
```

変数指定には値の変化に注目している変数名を列記する．

□

起動モードは 2 種類ある．

$ いつでもモード：計算の途中で一旦指定すると，計算環境のどれかの変数の値が変わったとき，いつでも関連する起動条件を評価する．

@ このときモード：起動文（@）で指定したときに，関連する起動条件を評価する．

起動文（aS_j）と起動の対象になる能動関数（aF_i）の関連は次のように定義される．

定義 3：起動文（aS_j）で起動の対象になる能動関数の集合（FS）：

$$FS = \{ aF_i \mid \exists x(x \in V_j \wedge x \in V_i) \wedge aC_i \}$$

V_i は起動条件 aC_i を構成している変数の集合であり，V_j は起動文 aS_j で指定された変数の集合である．

□

@モードでは，aS_j の変数リスト V_j 中に指定された変数（x）の値が新たに変化したかどうかを問わないで，この時点で対応する起動条件 aC_i の評価を行う．

例 1：素数の計算（@-mode）．

求めたい素数の個数（K）あるいは素数の値の範囲（PMAX）が指定されたとき，それを充たす素数を計算する．ただし，K はプログラムで扱える個数 KMAX 以下（K≤KMAX）とする．

得られた素数は配列（pr[last]，last<KMAX）に順次登録する．

素数の候補（cur）を既知の素数（pr[ind]）で割った剰余（imod）によって，3 つの起動条件を評価する．

この計算は，図 1 に示す関数で実現できる．

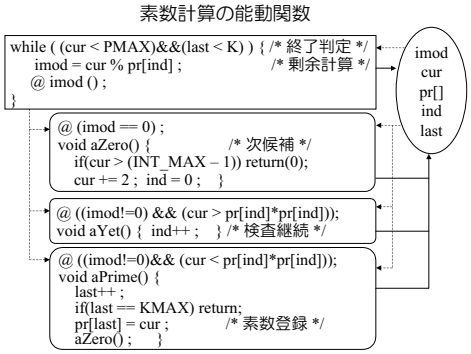

図 1: 素数を計算する能動関数

□

能動関数を直接呼び出すのではなく，値が更新された注目する変数（imod）を介して起動の切っ掛けを指示している．

3 状態遷移図

能動関数が，値を更新した変数や事象（event, 以下，イベント）の発生によって，間接的に起動することは，ソフトウェアシステムの構成の繋がりの柔らかさを意味し，変更や拡張を行ないやすくしている．

それには，ある処理によって，どのような更新やイベントが起きるのかを明瞭にしておくことが要点になる．計算状態の組を洗い出して，処理による状態の変化を定めることである．その結果，処理の流れを主にする表現とは違った，次のような状態の変化として表現できる．

(1) { 現状態，?条件，処理手順，!新条件，次状態 }

例 1a：素数の計算（@-mode）の状態遷移．

素数の計算（例 1）の状態の遷移は，図 2 に示す関数の繋がりとして実現できる．ここには，入力された計算要求が妥当かどうかの検査も含めている．

□

このような状態遷移の図を基にすると，機能の拡張を適切に指示できる．

例 1b：双子素数の計算の状態遷移．

素数の計算（例 1）に双子素数の抽出を追加する．図 2 の状態の遷移で，素数登録の状態に，図 3 のように双子判定の経路を付け加える．

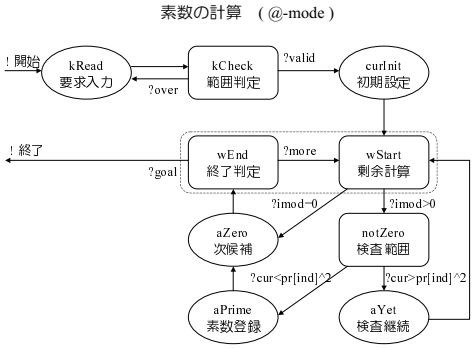

図 2: 素数計算の状態の遷移

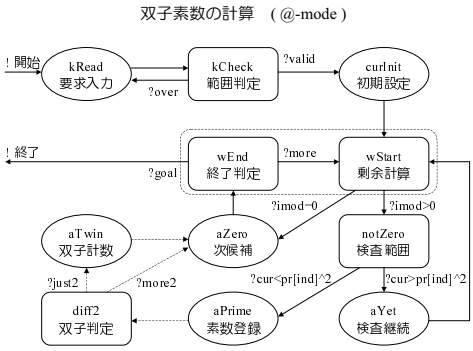

図 3: 双子素数の計算を追加

条件 ?just2 は一つ前の素数との差が 2 であること，条件 ?more2 は差が 2 より大きいことを示す．

□

起動条件は変数間の一般的な論理式で表現される（例 1）．逆に，条件となる論理式を，起動文側で評価して，それに対応してイベントを発生する構成の方法もある．

ソフトウェアが単一の構成要素（あるいは，ユニット unit）である場合は，前者が適しているが，複数のユニットで構成される場合は，後者のように単純なイベントを受理する形式が適している．

状態遷移の表現を次のように改める．

$$(2)\ (\mathrm{S1}\ ?\mathrm{ev1}) \rightarrow \{\mathrm{Process}()\} \rightarrow (!\mathrm{ev2}\ \mathrm{S2}^*)$$

状態（S1，S2）は，ソフトウェア全体の状態ではなく，あるユニットの状態を表す．また，$S2^*$ は，零個以上の状態の設定を表す．

これは，前条件（P）と後条件（R）による Hoare の記法 P{Q}R に対応しているが，前チェック（~P）や後チェック（~R）に当るイベントを生成することも含んでいる [2]．

例 2：観音開き自動扉の開閉（$-mode）．

人が近づいてきたことを感知する（?p）センサーと，それに対応して扉の開閉を制御するソフトウェアを構成する．この動作は，図 4 に示す状態遷移で表せる．

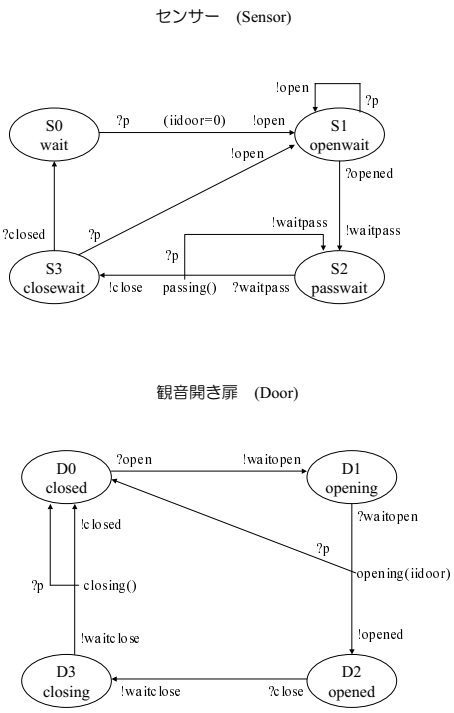

図 4: センサーと扉

たとえば，扉が閉じているとき（S3, D3）に，人が近づいてくる（?p）と，途中の状態から扉を開く．

```
activate :: state = D_23 & event = ev22
12::>>>>>>>>>>>>                        <<<<<<<<<<<<::
11::->>>>>>>>>>>>                      <<<<<<<<<<<<-::
10::-->>>>>>>>>>>>                    <<<<<<<<<<<<--::
 9::--->>>>>>>>>>>>                  <<<<<<<<<<<<---::
 8::---->>>>>>>>>>>>                <<<<<<<<<<<<----::
 7::----->>>>>>>>>>>>              <<<<<<<<<<<<-----::
^C --- !! enter into sig_intr() !!
activate :: state = D_21 & event = ev20
 7::----->>>>>>>>>>>>              <<<<<<<<<<<<-----::
 8::---->>>>>>>>>>>>                <<<<<<<<<<<<----::
 9::--->>>>>>>>>>>>                  <<<<<<<<<<<<---::
10::-->>>>>>>>>>>>                    <<<<<<<<<<<<--::
11::->>>>>>>>>>>>                      <<<<<<<<<<<<-::
12::>>>>>>>>>>>>                        <<<<<<<<<<<<::
```

人が近づいてきたことをキーボードからの割り込みで模擬して，イベント !p が発生したものとしている．

□

4 π 計算表示

図 4 のように，複数のユニット（センサーと扉）が，イベントを介して並行動作する様子は，状態遷移図だけでなく，π 計算式を用いて表現できる [4].

文献 [4] では，送信を port!()，受信を port?() で表現しているが，ここでは，引数を省いて，!event と ?event で表す．その意味では，π 計算式の一部しか使用しないが，本稿では π 計算式としておく．

イベントの発生 !event に対して，イベントの受信 ?event は 1 つとは限らない．

例 2p：自動扉の π 計算表示.

図 4 に示す状態遷移で表される動作を，π 計算式で表す．?q はシミュレーションの終了（quit）イベントである．

```
Auto Door System1 = Sensor | Door | ?q.0
Event1 = {{p, q}, {open, close, waitpass},
       {waitopen, opened, waitclose, closed}}
Sensor = initial.S0
   S0  = ?p.(iidoor = 0).!open.S1
   S1  = ?opened.!waitpass.S2
       + ?p.!open.S1
   S2  = ?waitpass.passing().!close.S3
       + ?p.!waitpass.S2
   S3  = ?closed.S0
       + ?p.!open.S1
Door   = initial.D0
   D0  = ?open.!waitopen.D1
   D1  = ?waitopen.opening(iidoor).!opened.D2
       + ?p.D0
   D2  = ?close.!waitclose.D3
   D3  = ?waitclose.closing().!closed.D0
       + ?p.D0
```

変数 iidoor は，扉の開き加減を表している．その値は例 2 の開閉状態の各行の数値（7..12）である．

□

π 計算式や状態遷移図を基にすると，機能や構成の変更や拡張を安全に遂行できる．それは，処理手順 Process() の置換え，イベントやユニットの追加などで行なえる．

例 2a：夜間のロック機構を付ける（Step2）.

夜間には扉は自動的に開かないようにする．ロック（lock）と認識機構（id-checker）を追加する．施錠・解放の時刻（?time）になると，変数 iLock を 1（L1 施錠状態）または 0（L0 解放状態）にする．施錠状態で，通行者の Id が有効（たとえば，奇数）ならば，イベント !IdOK によってロックを一時的に解放する（iLock=2，L2 一時解放状態）.

この動作は，図 5 に示すユニットの追加と，センサーの状態（S4）の追加で実現できる．

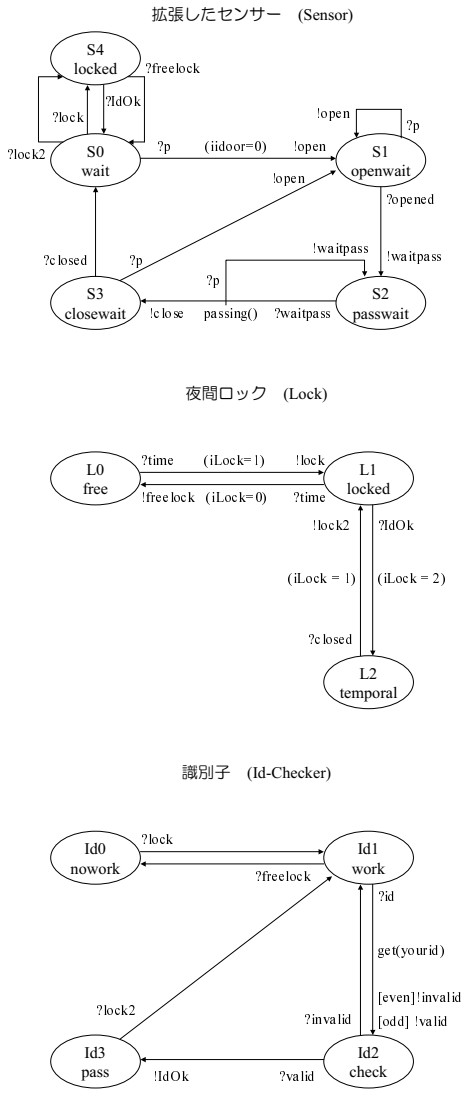

図 5: ロックと認識機構の追加

□

イベントを介したユニット間の関連図があると理解が深まるが，今のところ定めていない．

図 5 に対応する π 計算式は次のようになる．

例 2ap：追加部分の π 計算表示

```
Auto Door System2
   = Sensor | Door | Lock | IdChecker | ?q.0
Event2 = Event1 ∪
        {{time, id}, {lock, lock2, freelock}
         {valid, invalid, IdOK}}
Sensor = initial.S0
   S0  = ?p.(iidoor = 0).!open.S1
       + ?lock.S4
       + ?lock2.S4
   S4  = ?IdOk.S0
```

```
      + ?freelock.S0
   // S1, S2, S3 は System1 と同じ
Lock   = initial.L0
   L0  = ?time.(iLock = 1).!lock.L1
   L1  = ?time.(iLock = 0).!freelock.L0
       + ?IdOk.(iLock = 2).L2
   L2  = ?closed.(iLock = 1).!lock2.L1
IdChecker = initial.Id0
   Id0 = ?lock.Id1
   Id1 = ?freelock.Id0
       + ?id.get(yourid).
         ([(yourid % 2) == 0]!in-valid.
         +[else]!valid).Id2
   Id2 = ?invalid.Id1
       + ?valid.!IdOk.Id3
   Id3 = ?lock2.Id1
```

□

組込みシステムのソフトウェアや，LSI に載せるソフトウェアは，将来の変更や新しい機能への拡張に備えて，長期的な展望をもってユニット化しておくことが提唱されている [5].

これについて，状態遷移図やπ計算表示に基づいた能動関数によるソフトウェアの構成法が有効になる.

5 実行の形式

イベントをベースにした実行の形態は，次のサイクルの繰り返しになる.

1. 新しい次のイベント（?ev1）を取り出す.
2. （現状態 S1 と ?ev1）を起動条件とする能動関数を起動する.

これは，(2) 式の表現に対して，次の能動関数を起動することに対応する.

```
@ ((state == S1) && (newEvent == ev1) ) ;
type process()
{
  /* process 本体の手順 */
  state = S2;     /* 次の状態 S2 設定 */
  putEvent(ev2); /* イベントの生成 */
}
```

このようなソフトウェアの枠組み（framework）は，π計算表示から機械的に生成することができる．枠組みの中で本体の手順を詳細な記述で仕上げれば良い.

生成したイベント（! ev）を，イベントキューに貯めて，順に取り出して実行するサイクルは，このときモード（@-mode）よりも，いつでもモード（$-mode）に近い．それを次の形式で模擬実行する.

```
newEvent = getEvent(); /* ?ev1 */
@ newEvent () ;
```

例 1c：双子素数の計算をイベント表現する.

ユニットは 1 つであるが，図 3 を，論理条件ではなく，イベントによる実行サイクル形式に合わせ，単純なイベントの生成と取り出しを主にした解析をして，π計算式で表現する.

```
EventP   = {evGetK, evInit,   evCheck,
            evMod0, evCurEnd, evFutago}
Prime    = !evGetk.sSetting
               // by setInitialState()
sSetting = ?evGetk.aGetKandCheck().
         ( [i = 1]!evInit  // K の入力受理
         + [i = 2]!evInit  // PMAX の受理
         + [i = 0].0    // end of computation
         + [else ]!evGetk  // try input again
         ).sSetting
sSetting = ?evInit.primeInitial().
                  !evCheck.sComp
sComp    = ?evCheck.checkAndMod().
         ( [(cur<PMAX)&&(last<K)]
            (imod = cur % pr[ind]).
              ( [imod = 0]!evMod0
              + [imod > 0]!evCurEnd
              ).sComp
         + [else]!evGetk.sSetting
               // over computation
         )
sComp    = ?evMod0.aZero().!evCheck.sComp
               // next candidate
sComp    = ?evCurEnd.aYetorPrime().
         ( [cur > pr[ind]*pr[ind]]
            (ind++)!evCheck
         + [else]
            (last++).(登録)
                  !evFutago.!evMod0
         ).sComp
sComp    = ?evFutago.aFutagoNum().sComp
```

これは，図 3 での起動条件を，イベントの生成に置換えて，そのイベントの発生を起動条件にした形式（$-mode）になっている.

□

例 1b と例 1c で同じ計算結果を得るが，実行時間は次のように，3 倍弱に増している.

```
@-mode -- 例 1b : primetK6.act
your request 2 : get to 1000000100
to  100000000  #primes  5761455
               #futago   440312  (0.0764)
to 1000000000  #primes 50847534
               #futago  3424506  (0.0673)
We get    50847541  Prime number
```

```
We get pr[50847540] = 1000000097
          in sec01 =  50018.988 sec
```

括弧（）内の数値は，そこまでの素数に対する，双子素数の比率を表している．

```
$-mode -- 例 1c : autoPrime.act
to  100000000  #primes  5761455
               #futago   440312  (0.0764)
to 1000000000  #primes 50847534
               #futago  3424506  (0.0673)
            :: Time = 146125.607 sec
We get  50847541  Prime number
```

実行時間の違いは，$-mode（例 1c）では，論理条件の評価とイベントの判定の二重構成になっていること，起動文 @ newEvent に対応する起動条件の数が増えていること，および，実行サイクル数が増えていることに因る．

6 可変ハードウェアの構成

能動関数を用いたアクティブソフトウェアでは起動条件を評価して能動関数に起動を駆ける仕事が要点となる．これを，ハードウェア回路で実現すると，オーバヘッドを軽減できる．これに適した回路構成の一例を文献 [2] に挙げている．

本稿で提案する枠組では，(S1 & ev1）のような単純な起動条件を用いるため，一つの起動条件の判定をハードウェアで行うための回路が単純になると予測する．そのため，多くの起動条件の判定を回路上で並列に行うことができる．一般的な能動関数の起動条件の判定（たとえば，例 1，図 1）でハードウェアを用いる場合よりも，実行のオーバヘッドと実装の面積の点で，ハードウェアの効果が大きくなると期待できる．

状態名，イベント名を符号化して，必要最小のビット幅の一致回路を構成すると良い．ただし，拡張を考慮すると，ビット幅に余裕が要る．

起動条件を評価する回路はプログラムに対応して再構成される．イベントやユニットの追加・変更に伴って，回路の再構成も部分的に行なえると予測している．

例 2b：扉の開閉の不調を検出（Step3）．

扉を開閉するイベント（!open，!close）が発生してから，ある時間（たとえば，octime）経過しても開閉が終了しない（!opened，!closed が発生しない）場合には，扉の故障として警告イベント（!alarm）を出す．警告イベント（?alarm）をどのように扱うかは HT2 での処理に任せる．

時間の経過は図 6 の状態遷移をもつハードウェアタイマ（HT）で並列に計時する．

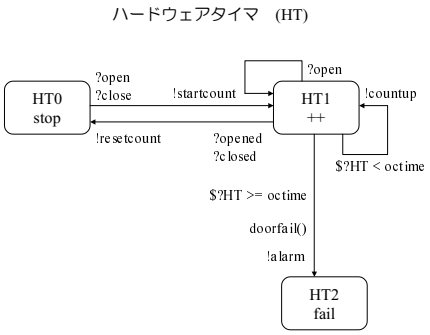

図 6: 扉の不調を検出する

タイマイベント（?HT >= octime）の検出は，$-mode を意識して，図 6 では $? で表している．

□

なお，人が近づいて来ても（例 2 の ?p を）検知しないというセンサの不調は，1 つのセンサだけでは検出できない．たとえば，2 つのセンサーをもつセンサユニットにして，一方が検出したとき，他方が検出していないことを，タイマを用いて判定するようにする．

7 能動関数の配列による構成

同じような操作をする能動関数は，配列に構成して一斉に独立して実行するようにできる [2]．

定義 1 を拡張した，能動関数の配列を導入する．

定義 1a：能動関数の配列 (AF)：

```
@SIZE (起動条件_aC) ;
type 能動関数名_AF(int ii, int size,
                   [仮引数の並び] )
{ 関数の本体 }
```

SIZE は能動関数の配列のサイズを表す変数の名前である．

□

関数の起動の際には，引数として，配列の添え字（ii）とサイズ（size）も受け取ることにする．

定義 2a：配列も対象にした起動文 (AS)：

```
@A 変数指定 ( [実引数の並び] );
```

起動モード（@）に記号 A を付けて，起動の対象に能動関数の配列があることを示す．

$モードのときも同じように $A となる．

□

例 3：長語整数の演算（加減乗算）．

たとえば，10 進数 7 桁を一語にして，多語の整数の演算を行なう．各数値は，属性として（桁数と語数）をもって表現する．また，行頭の数は，数値の番号（アドレス）を表している．

```
given operation  [20] a [21] to [22]   // 加算
20( 19,  3) =            12123 4567890 2005009.
21( 27,  4) =   999999 9987876 5678900 2005013.
22( 28,  4) :  1000000 0000000 0246790 4010022.

given operation  [22] s [20] to [23]   // 減算
22( 28,  4) =  1000000 0000000 0246790 4010022.
20( 19,  3) =            12123 4567890 2005009.
23( 27,  4) :   999999 9987876 5678900 2005013.

given operation  [20] m [21] to [24]   // 乗算
20( 19,  3) =            12123 4567890 2005009.
21( 27,  4) =   999999 9987876 5678900 2005013.
24( 46,  7) :     1212 3456774 3222595 5800399
            :          0618975 4844676 9110117.
```

□

これらの演算は，逐次には，下の桁から順次行なって結果を形成していく．

これを，能動関数の配列による構成にすると，計算の状態遷移が簡潔になる．

例 3a：能動関数の配列による加算の状態遷移．

たとえば，加算の場合，対応する各語の加算を独立に行なって，その語の和と桁上げを計算し（A1），下位からの桁上げがあれば，それを追加する（A2）．どの語の桁上げも発生しなければ加算は終了する．その状態遷移は図 7 のようになる．

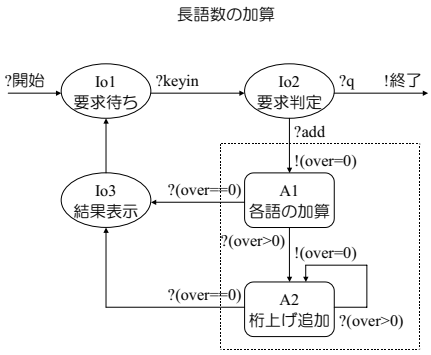

図 7: 多語整数加算の状態遷移

□

減算，乗算も加算と同じように構成できる．

配列のサイズは，加減算では語数の大きいオペランドに合わせるが，乗算では 2 つのオペランドの語数の和になる．

ここで，多語の計算（加減乗算）を，小数部をもつ実数へ拡張する．

例 3b：長語実数の演算（例 3 の拡張）．

小数部をもつ長語数の演算の一例を示す．数値の属性を（全桁数と語数，整数部の桁数と語数，小数部の桁数と語数）としている．

```
given operation  [20] a [21] to [22]  // 加算
20( 33, 5)[ 19, 3. 14, 2] =
             12123 4567890 2005009. 5556667 89
21( 34, 5)[ 27, 4.  7, 1] =
    999999 9987876 5678900 2005013. 50505
22( 42, 6)[ 28, 4. 14, 2] :
   1000000 0000000 0246790 4010023. 0607167 89,

given operation  [22] s [20] to [24]  // 減算
22( 42, 6)[ 28, 4. 14, 2] =
   1000000 0000000 0246790 4010023. 0607167 89
20( 33, 5)[ 19, 3. 14, 2] =
             12123 4567890 2005009. 5556667 89
24( 34, 5)[ 27, 4.  7, 1] :           // = [21]
    999999 9987876 5678900 2005013. 50505,

given operation  [20] m [21] to [25]  // 乗算
20( 33, 5)[ 19, 3. 14, 2] =
             12123 4567890 2005009. 5556667 89
21( 34, 5)[ 27, 4.  7, 1] =
    999999 9987876 5678900 2005013. 50505
25( 60, 9)[ 46, 7. 14, 2] :
      1212 3456774 3222595 6356065 8508361
           8477267 7362387. 2117027 6878445,
```

整数だけの長語から小数部をもつ実数への仕様の拡張に伴い次の変更を行なう．

1. 数値の入力と，出力表示の手順を変更する．

2. 加算の場合，例 3a では語数で行なっていた桁合せを，例 3b では整数部の語数で行なう．

   加算の流れの部分は，基本的には例 3a（図 7）と同じである．能動関数の中の一部が変更される．

3. 配列のサイズは，加減算の場合は，2 つのオペランドに対して，max（全語数）から，max（整数部の語数）+ max（小数部の語数）に変る．

□

逐次計算でのループの反復回数の変更は，能動関数の配列により，配列のサイズの変更になっている．

8 関連研究

アクティブソフトウェアに関連する概念として，オートノミックコンピューティング（autonomic computing）の構想が進められている [6]，[7]，[8].

文献 [6]，[7] では，これまでの計算システムとは，次の 4 つの観点が違っているとしている．

1. 自己構成（self-configuration）
2. 自己最適化（self-optimization）
3. 自己修復（self-healing）
4. 自己防御（self-protection）

文献 [8] では．次のように述べている．

“An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions.”

そして，適切な適応アルゴリズム（adaptive algorithm）を如何に創造するかの問題提起をしている．

一方，具体的なモデルとして，動的にソフトウェアを再構成するモデルの研究がある [9]．そこでは，システムの状態変数を用いて，分散システムの構造と行動を形式的に把握するモデルを提案している．キーポイントとして，現状の構成と再構成したものとの実行上の一貫性を挙げている．

いずれも，状況の変化に適応できるソフトウェア (adaptive software system) の構成を目指しているものである．

9 おわりに

アクティブソフトウェアを能動関数を用いて構成することにより,「自己を調整する」という特徴を実現することを目指している．そのために，π 計算表現 あるいは 状態遷移図を用いてソフトウェアを解析して，能動関数がどのように起動するかを明瞭に設計することを提案した．

一般には，複雑な論理式を起動条件にもつ能動関数を定義できる．しかし，複数のユニットからなるソフトウェアについては，単純なイベントを生成する形式にして，変更や拡張に該当する部分を明確にする方が良い．それにより，並列に動作するユニットの関係も判りやすくなる．

今後は，このような方針を基にして，π 計算表現から能動関数に関連する部分のプログラムの枠組みを生成して，自己調整を可能にする機構を考察しやすい環境を整えていく．

この場合，ソフトウェアの内部で生成されるイベントの変化だけでなく，計算環境から与えられるイベントを予測してどのような備えをしておくかが問題点である．

謝辞

本研究は，一部，日本学術振興会科学研究補助金基盤研究 (C)(2)15500023，および， 大川情報通信研究助成 (03-14) の支援による．

参考文献

[1] R. Laddaga, Active Software. In *Self-Adaptive Software, First International Workshop, IWSAS 2000*, LNCS 1936, Springer, 2000, pp. 11–19.

[2] 渡邉勝正, 井上晶広, 伴野 充, 蔵川 圭, 中西正樹, 山下茂, 能動関数によるアサーション検証設計. コンピュータソフトウェア, Vol. 22, No. 3 (2005), pp. 76–91.

[3] 渡邉勝正，井上晶広，山田 洋平，中西正樹，山下 茂, ソフトウェアの自己変更を支援する機構について. 信学技報, SS2004-34 (2004).

[4] R. Milner, Communicating and mobile systems : π-calculus. Cambridge University Press, 1999.

[5] 後藤康浩，勝つ工場．日本経済新聞社，2005，p.44.

[6] Jeffrey O. Kephart, David M. Chess, The Vision of autonomic Computing. IEEE Computer, Jan 2003, pp. 41–50.

[7] 福田剛志，オートノミックコンピューティング. 情報処理, Vol. 45, No. 4 (2004), pp. 348–353.

[8] IBM, Autonomic computing : IBM’s Perspective on the State of Information Technology. http://www-1.ibm.com/industries/government/doc/content/resource/thought/278606109.html.

[9] K. Whisnant, Z. T. Kalbarczyk and R. K. Iyer, A system model for dynamically reconfigurable software. IBM System Journal, Vol. 42, No. 1 (2003), pp. 45–59.