ＡＴＲテクニカルレポート表紙

〔公開〕

ＴＲ－Ｃ－００４８

要求理解プログラムの類似サービス検索部の一部についての ＡＲＴによる実現

横田　政憲
MASANORI YOKOTA

１９９０．４．１７

ＡＴＲ通信システム研究所

様式 3
(62.8 V=A)

ソフト登録票

登録番号	62ー	資産番号	

ソフト名	要求理解プログラムの類似サービス検索部の一部	
走行環境	ハード	Symbolics 他
	O S	
	使用リリース ・メモリ ・特殊i/o 等	
保守環境	作成言語	ART
	保存媒体 ・ファイル ・アクセス法 ・マニュアル 保管場所 等	clm05:>yokota>art-world のディレクトリの下 loader.lisp version1.art test4-kb.art test4-input.art

担当者	作成者	登録日	V/R
横田 政憲	横田 政憲		

処理概要

処理フロー(I/Oを含めて)

ART が乗っているsymbolics のART のウインドで clearメニューを選ぶ。

"clm05:>yokota>art-world>loader.lisp" をロードする。

ART のウインドで resetメニューを選ぶ。

"clm05:>yokota>art-world>test4-input.art" 中の assert で始まるリストを評価する。

（このファイル中には2つのリストがあるが、どちらか一方のみを評価する。）

ART のウインドで runメニューを選び、実行する。

(*) ART メニューの詳しい使い方については、ART マニュアルを参照のこと。

機能一覧

個条書きで

・入力は要求サービスを構成する動作フレーム群であるが、そのデータを基に類似サービスを推論によって検索し、入力の動作フレームと類似サービスを構成する動作フレームの間の対応関係を導く。

・類似サービスの検索は、入力と類似サービス中の動作フレームの一致の数によって類似度を判定し、類似度が最大のものを選んで推論を進める。

・推論はART のVIEW-POINT機構を用いた仮説推論によって行われ、出力結果は推論過程を表したVIEW-POINT構造であり、次の処理（入力と類似サービス中の動作フレームの類似の数による類似サービスの検索）に渡される。

注）外注ソフトの時は，固定資産番号および作成者に社名を記入する事。

要求理解プログラムの類似サービス検索部の一部についてのＡＲＴによる実現

目次

１．概要 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 1
２．データ構造 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 2
　2.1.　知識表現 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 2
　2.2.　推論制御のためのデータ構造 ・・・・・・・・・・・・・・・・・・・ 2
３．プログラムの概要 ・・・・・・・・・・・・・・・・・・・・・・・・・・ 3
　3.1.　入出力 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 3
　3.2.　アルゴリズムの概要 ・・・・・・・・・・・・・・・・・・・・・・・ 3
４．プログラムの詳細 ・・・・・・・・・・・・・・・・・・・・・・・・・・ 5
　4.1.　個々のルール ・・・・・・・・・・・・・・・・・・・・・・・・・・ 7
　4.2.　アルゴリズムとルールの対応 ・・・・・・・・・・・・・・・・・・２３
５．実行例 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・２４
６．今後の展開 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・２８
参考文献 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・２８
付録１　プログラム ・・・・・・・・・・・・・・・・・・・・・・・・・・２９
付録２　知識ベースの例 ・・・・・・・・・・・・・・・・・・・・・・・・４０
付録３　入力例 ・・・・・・・・・・・・・・・・・・・・・・・・・・・・４５
付録４　今後の拡張（未完成プログラム） ・・・・・・・・・・・・・・・・４６

要求理解プログラムの類似サービス検索部の一部についてのＡＲＴによる実現

１９９０年4月１７日　横田　政憲

１．概要

類似サービス検索部は、要求理解プログラムの一部であり、マッチング部で得られた結果をもとに類似サービスを検索するプログラムである。処理の概要は、参考文献［１］を参照されたい。

入出力は以下の通りである。

［入力］　要求サービスとそれを構成する入力動作フレーム群。要求サービスは、parts スロットにより、各入力動作フレームとの構成関係が規定されている。また、各入力動作フレームは、match スロットにより、知識ベース中の動作フレームとリンクが張られており、これらの時間順序関係は、pre-act, post-act スロットによって、規定されているものとする。

［出力］　出力の候補は複数ある可能性があるので、それぞれの候補に対し、view-point (view-point については参考文献［２］を参照のこと）を割り当てる。出力の情報としては、類似サービス名及び、各入力動作フレームと対応する類似サービス中の動作である。類似サービス名の格納場所は world-controle-factorフレームの similar-serviceスロットである。入力動作フレームと対応する動作は対応の状態が３通りあり、それぞれ以下に示す場所に格納される。

・入力動作フレームが対応する動作と一致している場合
　入力動作フレームの matchスロットに格納。この場合、similar-act スロットの値には何も入っていない。

・入力動作フレームが対応する動作と類似している場合
　入力動作フレームの similar-actスロットに格納。

・入力動作フレームと対応する動作がない場合
　入力動作フレームの similar-actスロットに特別な値 "*"を格納。

類似サービス検索部は、以下の３つの部分よりなる。

(1)入力動作フレームと類似サービス中の動作の一致による類似サービスの検索。
(2)入力動作フレームと類似サービス中の動作の類似による類似サービスの検索。
(3)出力のための後処理。

現在のところ(1)がＡＲＴにより実現しているので、その実現手法について報告する。

2．データ構造

この節では、本プログラムで用いるスロット、推論制御のためのスロットについて説明する。

2.1. 知識表現

以下に知識ベースを構成するフレーム及び入力フレームのためのスロットを示す。これは、本プログラムで用いた英語名のスロットであり、文献［1］で述べた日本語名のスロットとの対応関係を示す。それぞれの意味については、文献［1］を参照されたい。

・ako : 上位
・kind : 下位
・part-of : 全体
・parts : 部分
・inverse : 逆関係
・inverse-act : 逆動作
・input-act : 入力動作
・output-act :出力動作
・agent : 主体
・object : 対象
・condition : 条件
・change-from : 事前
・change-to : 事後
・message-from : 始点
・message-to : 終点
・intention : 意図
・media : メディア
・form : 形式
・has-a : 所有
・state : 状態
・role : 役割
・match : マッチ
・post-act : 後動作
・pre-act : 前動作
・match-intention : 類似

2.2. 推論制御のためのデータ構造

推論を制御するためのフレームとして次のフレームとスロットを用いる。

(a) control-factor — 推論全体を制御するためのフレーム。
・act-number : 入力動作フレームの数
・max-match-factor : 類似サービス中の動作と入力動作の一致の数の最大値。
・match-phase : マッチングプログラムの全体としてのフェーズ
・whole-act : 類似サービスの候補

(b) world-control-factor — 各世界における推論を制御するためのフレーム。
・match-factor : 類似サービス中の動作と入力動作フレームの一致の数。
・similar-service : 類似サービスの仮説
・partial-action : 入力動作フレームとの一致をまだ調べていない部分動作
・performing-act : シミュレート中の類似サービス中の動作。
・view-act : 部分動作との一致を調べようとしている入力動作フレーム
・next-act : view-actの次にくるべき入力動作フレーム
・p-init : 部分動作に展開したとき、最初に起こる動作
・macting-phase : 推論の段階を管理する情報

(c) 推論制御のための入力動作フレームのスロット
・whole-act : 入力動作フレームを部分とするような知識ベース中の動作。
・match-as-part : 類似サービス中の動作で、入力動作が一致する、或いは、その動作の部分動作と一致するような動作。

(d) 推論制御のための知識ベース中の動作フレームのスロット
・link : ある動作を部分動作に展開したとき、部分動作から見た全体の動作。
・status : 動作が実行されたかどうかを示す情報。

3．プログラムの概要

類似サービスを仮定し、その部分動作を実行をシミュレートする。この過程で、入力動作フレームと実行中の部分動作の一致を判定し、最も一致の多い組み合わせを類似サービスとして検索する。ここで、動作の実行とは、condition，change-from，change-to スロットの情報を基に、状態を変化させていくことである。

3．1．入出力

［入力］

・フレーム ： 要求サービスのフレーム、入力動作フレーム群

・要求サービスのスロット ： parts

・入力動作フレーム群のスロット ： match、pre-act、post-act

・control-factorのスロット ： act-number

入力の例に関しては付録3を参考のこと。

［出力］

推論過程のview-point構造。但し、ルール"make-result-world" に弱冠の修正を加え、ルール"poison-unrelated-world"をアクティブにすることで、結果のみのview-pointを作成することも可能。

推論結果はその推論を導いたルールが発火した時点のview-pointで作成され、それ以降のview-pointに継承される。以下に推論結果の格納場所を示す。

・類似サービス — world-controle-factor フレームの similar-serviceスロット

・入力動作フレームと一致する動作 — 各入力フレームのmatch-as-part スロット

　但し、一致する動作がない場合は値は入っていない。

入出力の例については、5章で説明する。

3．2．アルゴリズムの概要

このアルゴリズムでは次の4種類の仮説を用いる。

(仮説1) 類似サービスの仮説。

(仮説2) 部分動作への展開と部分動作間の順序に関する仮説。

(仮説3) ある入力動作が類似サービス中の動作と一致するかしないかという仮説。

(仮説4) ある入力動作が類似サービス中のどの動作と一致するかという仮説。

これらの仮説を制御するために、次のデータを定義する。

(1) 仮説世界を制御するフレーム(world-control-factor)のスロット

(a) similar-service = SS : 類似サービスの仮説。

(b) performing-act = PA : シミュレート中の類似サービス中の動作。

(c) view-act = VA : 一致を試みようとする入力動作。

(d) next-act = NA : view-actの次にくるべき動作。仮説3はNAスロットに値が入った時点で行う。

(e) match-factor = MF : 類似サービス中の動作と入力動作の一致の数。シミュレートの開始段階では一致する可能性のある全ての入力動作の数。仮説3により、1づつ減っていく。

(2) 推論制御のための入力動作のスロット

(f) match-as-part = MAP : 類似サービス中の動作で、入力動作が一致する、或いは、その動作の部分動作と一致するような動作。

以下にアルゴリズムの概要を示す。Step3 以降は、MFの値の最も大きい仮説世界を優先して推論を進める。

[Step1] 類似サービスの仮定 (仮説1)。

・(新しい仮説世界で) SS <--- 類似サービス, PA <--- 類似サービス.

[Step2] 個々の入力動作に対し、類似サービス中の動作と一致する可能性があれば MAP スロットに類似サービスを代入。MAP スロットに値SSを持つ入力動作の数をMFに代入。

・MAP(入力動作n) <--- SS, …, MF <--- MAP スロットに値を持つ入力動作の数.

[Step3] 類似サービスを実行するための状態を作成。

・類似サービスのcondition スロットにより状態を作成.

[Step4] 入力動作の先頭動作をNAに代入。

・NA <--- 入力動作の先頭.

[Step5] MAP(NA) = nil ならば、NAの次の動作をNAに代入。Step5 へ。

・if MAP(NA) = nil, then NA <--- post-act(NA), go Step5.

[Step6] NAが類似サービス中の動作と一致しないという仮定 (仮説3)。
一致するという仮定を行った場合は、NAをVAに代入。

・(新しい仮説世界で) MAP(NA) <--- nil, MF <--- MF - 1, go Step5.

・(新しい仮説世界で) VA <--- NA, NA <--- nil.

[Step7] VAの次の動作をNAに代入。VAの次の動作がない場合はStep16 へ。

・ if post-act(VA) = nil, then go Step 16, else NA <--- post-act(VA).

[Step8] MAP(NA) = nil ならば、NAの次の動作をNAに代入。Step8 へ。
NAの次の動作が無い場合は、Step16 へ。

・if MAP(NA) = nil,
then if post-act(NA) = nil, then NA <--- nil, go Step16,
else NA <--- post-act(NA), go Step8.

[Step9] NAが類似サービス中の動作と一致しないと仮定 (仮説3)。一致すると仮定する場合は、PA = SS のならばStep 12へ。それ以外はStep10 へ。

・(新しい仮説世界で) MAP(NA) <--- nil, MF <--- MF - 1, go Step8.

・ if PA = SS, then go Step 12, else go Step 10.

[Step10] まだ実行していない類似サービス中の動作の中で、NAを部分とするような動作を検索、NAのMAP スロットに代入 (仮説4)。

(注. NAを部分とする動作がない場合、ルールが走らず仮説世界は凍結される。)

・(新しい仮説世界で) MAP(NA) <--- NAを部分とするような動作.

[Step11] MAP(VA) = PA の場合、MAP(NA) = PA ならば、Step12 へ。
MAP(NA) ≠ PA ならば、Step13 へ。
MAP(VA) ≠ PA の場合、MAP(NA) = PA ならば、Step14 へ。
MAP(NA) ≠ PA ならば、Step15 へ。

[Step12] PAを部分動作に展開し、その中で、VA,NA を部分とするような動作を仮定し、

それぞれの MAPスロットにその動作を代入（仮説4）、部分動作の先頭の動作を

PAに代入（仮説2 ：展開のしかた、及び、先頭の動作の仮定）、Step11 へ。

（注．PAが部分動作に展開できない場合、ルールが走らず仮説世界は凍結される。）

・（新しい仮説世界で）MAP(VA) <--- PA の部分動作でVAを部分とするような動作,

MAP(NA) <--- PA の部分動作でNAを部分とするような動作,

PA <--- PAの部分動作の先頭動作，go Step11.

[Step13] VAとPAが一致したので、次の動作を比較する。NAをVAに代入。Step7 へ。

・ VA <--- NA, NA <--- nil, go Step7.

[Step14] 順序の制約により、この仮説世界を抹消。

[Step15] PAを実行、次の動作をPAに代入（仮説2 ：次の動作の仮定）、Step11 へ。

（注．PAの次の動作がない場合、ルールが走らず仮説世界は凍結される。）

・ PA を実行，（新しい仮説世界で）PA <--- PAの次の動作，go Step 11.

[Step16] PAを実行、次の動作をPAに代入（仮説2 ：次の動作の仮定）、Step16 へ。

（注．類似サービス中の全ての動作が実行されれば終了。

PAの次の動作がない場合は、ルールが走らず仮説世界は凍結される。）

・ PA を実行，if 類似サービス中の全ての動作が実行，then 終了，

else（新しい仮説世界で）PA <--- PAの次の動作，go Step 16.

4．プログラムの詳細

図１にプログラムの流れを示す。ここで記号の意味は次の通りである。

[r.数]　：新しく仮説世界を作らないルールを表す。

[r.数]（網掛け）　：新しく仮説世界を作るルールを表す。

(s.数)　：アクティブな状態を表す。

(s.数)（網掛け）　：アクティブでない状態を表す。この状態は control-factor フレームの max-match-factor スロット値によってこの状態はアクティブになる。

(dead.state)　：これ以上先に進まない状態を表す。

<c.数>, △c.数　：条件分岐を表す。

○ ：　複数の仮説世界を作成するルールが発火することを示す。○から同一ルールへ一本のパスしか出ていない場合は、そのルールによっては一つの仮説世界しか作成されないことを意味し、複数のパスが出ている場合は、同一ルールが複数回発火することによって複数の仮説世界が作成されることを意味する。

以下に個々のルールについて説明する。

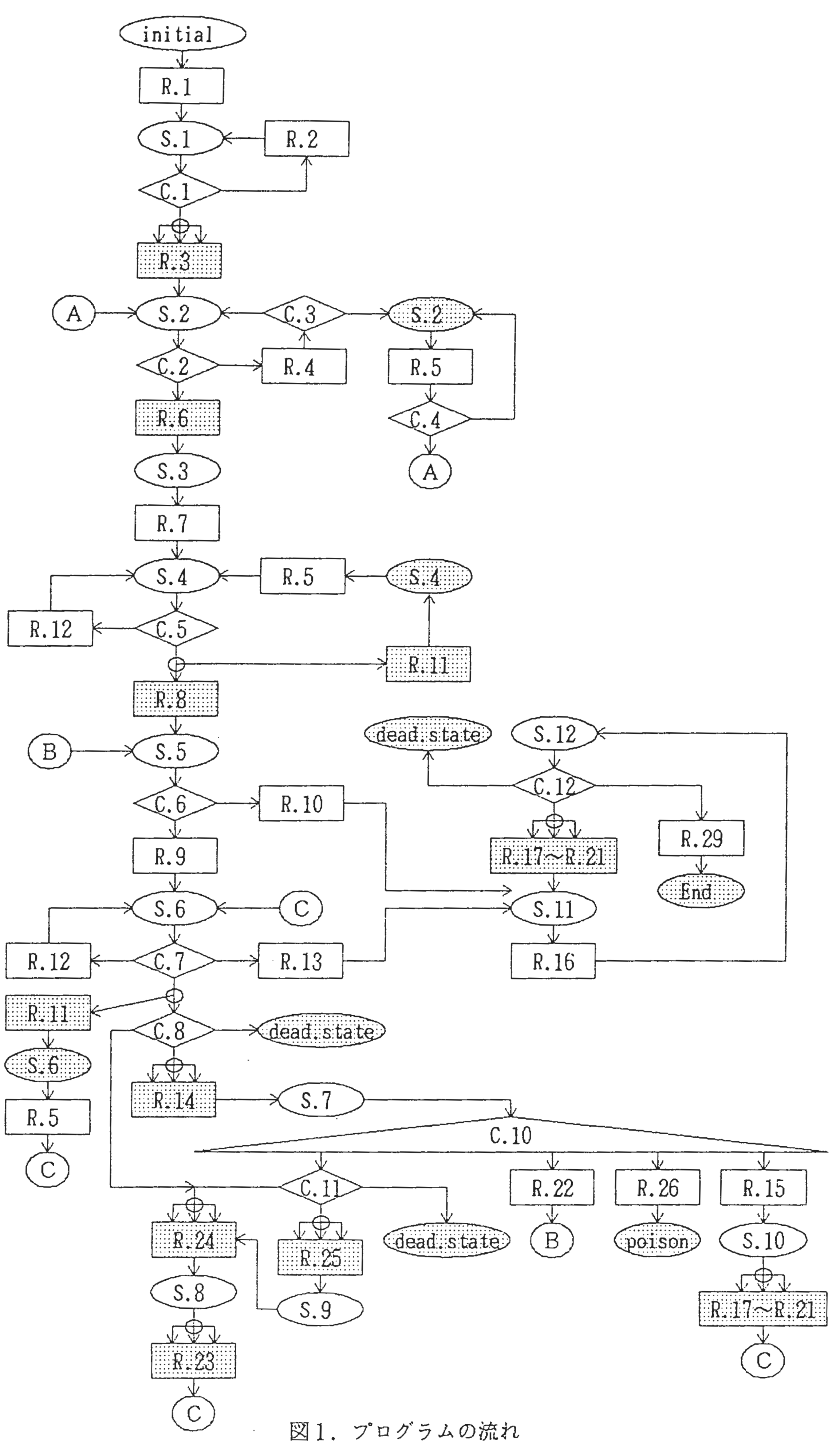

図1．プログラムの流れ

4.1. 個々のルール

R.1 : search-whole-action-mode

```
(defrule search-whole-action-mode "control-fact-for-initial"
  (declare (salience 1000))
  (viewpoint ?VP
             (schema control-factor
                     (act-number ?AN)
                     (not (max-match-factor ?))))
  =>
  (at ?VP (assert (max-match-factor control-factor ?AN)
                  (matching-phase world-control-factor search-whole-action))))
```

プログラムが始動するためのルール。入力動作フレームの数を、max-match-factorに入れ、matching-phaseをsearch-whole-action にする。

R.2 : search-whole

```
(defrule search-whole "search-whole-action"
  (declare (salience 100))
  (viewpoint ?VP
             (matching-phase world-control-factor search-whole-action)
             (or (match ?inp ?act)
                 (whole-act ?inp ?act))
             (or (part-of ?act ?whole-act)
                 (and (or (ako ?act ?act1)
                          (ako ?act1 ?act))
                      (part-of ?act1 ?whole-act))))
  =>
  (at ?VP (assert (whole-act ?inp ?whole-act)
                  (whole-act world-control-factor ?whole-act))))
```

ある入力動作フレームのmatch スロットの値から、part-of, akoスロットによるリンクを辿っていくことにより、入力動作フレームが部分になりえるような動作を検索し、その入力動作フレームのwhole-act スロットに入れる。この動作は、類似サービスの候補になりえるので、world-control-factorのwhole-act にも入れる。

このルールは、次に走る類似サービスを仮定するルールより優先順位が高いので、このような動作を全て検索しつくすまで発火しつづける。

R.3 : similar-service-hypotheses

```
(DEFRULE similar-service-hypotheses "make hypotheses"
  (declare (salience -100))
  (viewpoint ?VP
           (schema world-control-factor
                   ?fact <- (matching-phase search-whole-action)
                   (whole-act ?act)))
  =>
  (at ?VP (sprout (retract ?fact)
                 (assert (schema world-control-factor
                                 (similar-service ?act)
                                 (match-factor 0)
                                 (partial-action ?act)
                                 (matching-phase phase-0))))))
```

類似サービスを仮定し、similar-service に入れる。初期値として、match-factorに0を入れ、matching-phaseをphase-0 にする。

このルールは類似サービスの候補のそれぞれに対して複数回発火し、それぞれに対し異なった仮説世界を生成する。アルゴリズムのStep1 に対応。

R.4 : set-match-as-part

```
(defrule set-match-as-part "set match-as-part slot of act"
  (declare (salience -110))
  (viewpoint ?vp
           (schema world-control-factor
                   (matching-phase phase-0)
                   (similar-service ?act)
                   ?fact <- (match-factor ?factor))
           (schema ?input-act
                   (match ?)
                   (whole-act ?act))
           (not (match-as-part ?input-act ?)))
  =>
  (at ?vp
      (retract ?fact)
      (assert (match-as-part ?input-act ?act)
              (match-factor world-control-factor =(+ ?factor 1)))))
```

各入力動作フレームが類似サービスの部分になりえるかどうかを調べ、なりえるならばmatch-as-part スロットに類似サービスを入れ、match-factorを＋１する。

このルールは、次に走るルールより優先順位が高いので、全ての入力動作フレームについて調べるまで発火しつづける。アルゴリズムのStep2 に対応。

R.5 : change-control-factor

```
(defrule change-control-factor ""
  (declare (salience -200000))
  (viewpoint ?VP
            ?fact <- (max-match-factor control-factor ?max-factor)
            (not (match-phase control-factor ?)))
  =>
  (at ?VP
     (retract ?fact)
     (assert (max-match-factor control-factor =(- ?max-factor 1)))))
```

match-factorの最大値であるmax-match-factorを制御するためのルール。

このルールの優先度は最も低く、他の全てのルールが発火しなくなった時点、即ち、全ての状態がアクティブでなくなった時点で発火する。このルールの発火により、S.4, S.6状態がアクティブになり、S.2 状態はmatch-factorの値によってアクティブになりえる。

R.6 : set-initial-state

```
(defrule set-initial-state "set initial state"
  (declare (salience -120))
  (viewpoint ?VP
            (schema world-control-factor
                   ?fact <- (matching-phase phase-0)
                   (similar-service ?act)
                   (match-factor ?factor))
            (max-match-factor control-factor
                             ?max-factor&:(?max-factor = ?factor)))
  =>
  (at ?VP (sprout (retract ?fact)
                 (dolist (cond (list*$ (get-schema-value ?Act 'condition)))
                   (put-schema-value (first (list*$ cond))
                                    (second (list*$ cond))
                                    (third (list*$ cond))))
                 (assert (schema world-control-factor
                                (matching-phase Phase-1)
                                (performing-act ?act))))))
```

類似サービスを実行するための状態を作成。matching-phaseをPhase-1 にし、類似サービスをperforming-actに入れる。アルゴリズムのStep3 に対応。

R.7 : set-initial-next-act

```
(defrule set-initial-next-act "If VA=nil, NA=nil, set NA"
  (declare (salience -200))
  (viewpoint ?VP
            (schema world-control-factor
                    (matching-phase Phase-1)
                    (not (view-act ?))
                    (not (next-act ?)))
            (match ?act ?)
            (not (pre-act ?act ?)))
  =>
  (at ?VP (assert (next-act world-control-factor ?act))))
```

入力動作フレームの先頭をnext-actに入れる。アルゴリズムのStep4 に対応。

R.8 : set-initial-view-act

```
(defrule set-initial-view-act "If VA=nil, NA=t, map(NA)=similar-service,
                                VA <- NA, NA<- nil"
  (declare (salience -200))
  (viewpoint ?VP
            (schema world-control-factor
                    (not (view-act ?))
                    ?fact <- (next-act ?NA)
                    (similar-service ?map))
            (match-as-part ?NA ?map))
  =>
  (at ?VP (sprout
           (retract ?fact)
           (assert (view-act world-control-factor ?NA)))))
```

状態S.4 で、next-actのmatch-as-part スロットがsimilar-service であるとき、next-actが類似サービス中の動作と一致すると仮定した場合、next-actをview-actに入れる。アルゴリズムのStep6 に対応。

R.9 : set-next-act

```
(defrule set-next-act "If VA=t, NA=nil, NA <- post(VA)"
  (declare (salience -200))
  (viewpoint ?VP
            (schema world-control-factor
                    (matching-phase Phase-1)
                    (match-factor ?factor)
                    (view-act ?VA)
                    (not (next-act ?)))
            (max-match-factor control-factor
                              ?max-factor&:(?max-factor = ?factor))
            (post-act ?VA ?post))
  =>
  (at ?VP (assert (next-act world-control-factor ?post))))
```

状態S.5 で、view-actの次の動作をnext-actに入れる。アルゴリズムのStep7 に対応。

R.10 : cant-set-next-act

```
(defrule cant-set-next-act "If VA=t, NA=nil, post(VA)=nil, change mode"
  (declare (salience -200))
  (viewpoint ?VP
             (schema world-control-factor
                     ?fact <- (matching-phase Phase-1)
                     (view-act ?VA)
                     (not (next-act ?)))
             (not (post-act ?VA ?)))
  =>
  (at ?VP
      (retract ?fact)
      (assert (matching-phase world-control-factor Phase-2))))
```

状態S.5 でview-actの次の動作がない場合、即ち、view-actが入力動作フレームの最後である場合、matching-phaseをPhase-2 にする。アルゴリズムのStep7 に対応。

R.11 : hyp-next-act-inadequate

```
(defrule hyp-next-act-inadequate "If NA=similar-service, map(NA)=t,
                                   then hyp map(NA)=nil"
  (declare (salience -190))
  (viewpoint ?VP
             (schema world-control-factor
                     (next-act ?NA)
                     (similar-service ?SS)
                     ?fact1 <- (match-factor ?factor))
             (max-match-factor control-factor
                               ?max-factor&:(?max-factor = ?factor))
             ?fact2 <- (match-as-part ?NA ?SS))
  =>
  (at ?VP (sprout
           (retract ?fact1 ?fact2)
           (assert (match-factor world-control-factor =(- ?factor 1))))))
```

状態S.4, S.6でnext-actが類似サービス中の動作と一致しないと仮定する。このとき、next-actのmatch-as-part がnil になり、match-factorが１減る。アルゴリズムのStep6 とStep9 に対応。

R.12 : retract-inadequate-next-act

```
(defrule retract-inadequate-next-act
         "If map(NA)=nil, NA <- post(NA)"
  (declare (salience -190))
  (viewpoint ?VP
             (max-match-factor control-factor ?max-factor)
             (match-factor world-control-factor
                           ?max-factor&:(?max-factor = ?factor))
            ?fact <- (next-act world-control-factor ?NA)
             (not (match-as-part ?NA ?))
             (post-act ?NA ?PA))
;;  (similar-service world-control-factor ?)
  =>
  (at ?VP
      (retract ?fact)
      (assert (next-act world-control-factor ?PA))))
```

状態S.4, S.6でnext-actのmatch-as-part スロットがnil のとき、next-actの次の動作をnext-actに代入する。アルゴリズムのStep5 とStep8 に対応。

R.13 : retract-inadequate-next-act-2

```
(defrule retract-inadequate-next-act-2
        "If map(NA)=nil, post(NA)=nil, Change Mode"
  (declare (salience -200))
  (viewpoint ?VP
            (max-match-factor control-factor ?max-factor)
            (schema world-control-factor
                   (match-factor ?factor&:(?factor = ?max-factor))
                   ?fact <- (matching-phase Phase-1)
                   ?fact1 <- (next-act ?NA))
            (not (match-as-part ?NA ?))
            (not (post-act ?NA ?)))
  =>
  (at ?VP
     (retract ?fact ?fact1)
     (assert (matching-phase world-control-factor Phase-2))))
```

状態S.6 でnext-actのmatch-as-part スロットがnil で、next-actの次の動作がない、即ちnext-actが入力動作フレームの最後である場合、matching-phaseをPhase-2 にする。アルゴリズムのStep8 に対応。

R.14 : change-map-of-next-act

```
(defrule change-map-of-next-act
            "If map(NA)=similar-service, whole(NA) == (rest or PA),
             then (sprout), map(NA) <- (rest or PA)"
  (declare (salience -200))
  (viewpoint ?VP
            (schema world-control-factor
                   (next-act ?NA)
                   (similar-service ?map))
            ?fact <- (match-as-part ?NA ?map)
            (or (whole-act ?NA ?whole& ?map)
                (match ?NA ?whole)
                (and (match ?NA ?x)
                     (or (ako ?x ?whole)
                         (ako ?whole ?x))))
            (or (performing-act world-control-factor ?whole)
                (REST-ACT ? ?whole)))
  =>
  (at ?VP (sprout (retract ?fact)
          (assert (match-as-part ?NA ?whole)))))
```

状態S.6 で、next-actのmatch-as-part スロットがsimilar-service の場合、まだ実行していない類似サービス中の動作の中で、next-actを部分とするあるいは一致するような動作を検索し、next-actのmatch-as-part スロットに入れる。

ここで、whole-act スロットの値は入力動作フレームを部分とするような動作であり、REST-ACTは動作を分割したときにまだ実行していない動作である。

アルゴリズムのStep10に対応。

R.15 : perform-act

```
(defrule perform-act "If PA <> map(VA), PA<> map(NA),
                      then perform PA, status(PA) <- done"
  (declare (salience -210))
  (viewpoint ?VP
           (schema world-control-factor
                  (performing-act ?PA)
                  (view-act ?VA)
                  (next-act ?NA)
                  (similar-service ?SS))
           (not (status ?PA done))
           (not (match-as-part ?VA ?PA))
           (not (match-as-part ?NA ?PA))
           (not (match-as-part ?NA ?SS))
           (match-as-part ?NA ?))
  =>
  (at ?VP
     (dolist (from (list*$ (get-schema-value ?PA 'change-from)))
       (retract-schema-value (first (list*$ from))
                             (second (list*$ from)) nil
                              (third (list*$ from))))
     (dolist (to (list*$ (get-schema-value ?PA 'change-to)))
       (put-schema-value (first (list*$ to)) (second (list*$ to))
                         (third (list*$ to))))
     (assert (status ?PA done))))
```

状態S.7 で、view-act、next-actのmatch-as-part スロットが共にperforming-actと一致しない場合、performing-actを実行する。performing-actの実行は、change-from スロットの値を取り除き、change-to スロットの値を挿入することで行われる。また、実行したことを示すために、statusをdoneにする。

アルゴリズムのStep11, Step15に対応。

R.16 : perform-act-after-match

```
(defrule perform-act-after-match ""
  (declare (salience -210))
  (viewpoint ?VP
           (schema world-control-factor
                  (matching-phase Phase-2)
                  (performing-act ?PA)
                  (view-act ?)
                  (not (next-act ?))))
  =>
  (at ?VP
     (dolist (from (list*$ (get-schema-value ?PA 'change-from)))
       (retract-schema-value (first (list*$ from))
                             (second (list*$ from)) nil
                             (third (list*$ from))))
     (dolist (to (list*$ (get-schema-value ?PA 'change-to)))
       (put-schema-value (first (list*$ to)) (second (list*$ to))
                         (third (list*$ to))))
     (assert (status ?PA done))))
```

状態S.11でperforming-actを実行する。performing-actの実行は、R.15と同じ。

アルゴリズムのStep16に対応。

R.17 : next-perform-act-1

```
(defrule next-perform-act-1
        "If status(PA)=done, ako(PA)=composite-act, then search PA"
  (declare (salience -210))
  (viewpoint ?VP
            ?fact <- (performing-act world-control-factor ?PA)
            ?fact1 <- (status ?PA done)
            ?fact2 <- (link ?PA ?whole)
            ?fact3 <- (rest-act ?whole ?rest)
            (ako ?PA composite-act)
            (not (and (condition ?rest (?name ?slot ?value))
                      (not (schema ?name
                                   (?slot ?value)))
                      (not (and (ako ?ako-name ?name)
                                (schema ?ako-name
                                        (?slot ?value))))
                      (not (and (ako ?ako-value ?value)
                                (schema ?name
                                        (?slot ?ako-value))))
                      (not (and (ako ?ako-name ?name)
                                (ako ?ako-value ?value)
                                (schema ?ako-name
                                        (?slot ?ako-value)))))))
 =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))
```

状態S.10, S.12で、実行したperforming-actがcomposite-act である場合、次の動作を検索し、performing-actに代入する。ここで、linkスロットは実行中の動作とその全体である動作を結びつけ、rest-actスロットはまだ実行していない動作を示す。次の動作は複数の候補がある可能性があるので、このルールは新しい仮説世界を生成する。

アルゴリズムのStep15, Step16に対応。

R.18 : next-perform-act-2

```
(defrule next-perform-act-2
        "If status(PA)=done, ako(PA)=function, then PA <- output-Act(PA)"
  (declare (salience -210))
  (viewpoint ?VP
            ?fact <- (performing-act world-control-factor ?PA)
            ?fact1 <- (status ?PA done)
            ?fact2 <- (link ?PA ?whole)
            ?fact3 <- (rest-act ?whole ?rest)
            (ako ?PA functional-act)
            (or (output-act ?PA ?rest)
                (and (output-act ?PA ?x)
                     (or (ako ?x ?rest)
                         (ako ?rest ?x)))))
 =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))
```

R.17と同様であるが、実行したperforming-actがfunctional-actである場合。

R.19 : next-perform-act-3

```
(defrule next-perform-act-3
        "If status(PA)=done, ako(PA)=send-message,
             then PA <- inverse-act(PA)"
  (declare (salience -210))
  (viewpoint ?VP
             ?fact <- (performing-act world-control-factor ?PA)
             ?fact1 <- (status ?PA done)
             ?fact2 <- (link ?PA ?whole)
             ?fact3 <- (rest-act ?whole ?rest)
             (ako ?PA send-message)
             (or (inverse-act ?PA ?rest)
                 (and (inverse-act ?PA ?x)
                      (or (ako ?x ?rest)
                          (ako ?rest ?x)))))
 =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))
```

R.17と同様であるが、実行したperforming-actがsend-messageである場合。

R.20 : next-perform-act-4

```
(defrule next-perform-act-4
        "If status(PA)=done, ako(PA)=receive-massage, input(X)=PA,
            then PA <- X "
  (declare (salience -210))
  (viewpoint ?VP
             ?fact <- (performing-act world-control-factor ?PA)
             ?fact1 <- (status ?PA done)
             ?fact2 <- (link ?PA ?whole)
             ?fact3 <- (rest-act ?whole ?rest)
             (ako ?PA receive-message)
             (or (input-act ?rest ?PA)
                 (and (input-act ?x ?PA)
                      (or (ako ?x ?rest)
                          (ako ?rest ?x)))))
 =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))
```

R.17と同様であるが、実行したperforming-actがreceive-massage である場合。

R.21 : next-perform-act-5

```
(defrule next-perform-act-5
        "If status(PA)=done, rest(whole-act(PA)):nil,
            then PA <- whole-act(PA)"
  (declare (salience -210))
  (viewpoint ?VP
            ?fact <- (performing-act world-control-factor ?PA)
            ?fact1 <- (status ?PA done)
            ?fact2 <- (link ?PA ?whole)
            (not (rest-act ?whole ?)))
  =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2)
                (assert (performing-act world-control-factor ?whole)))))
```

R.17と同様であるが、実行した動作がその全体となる動作の最後の動作である場合、その全体となる動作をperforming-actに代入する。

R.22 : next-action

```
(defrule next-action "If PA=map(VA), PA<>map(NA), then VA <- NA, NA <- nil"
  (declare (salience -210))
  (viewpoint ?VP
            (schema world-control-factor
                    (performing-act ?PA)
                    ?fact1 <- (view-act ?VA)
                    ?fact2 <- (next-act ?NA)
                    (similar-service ?SS))
            (match-as-part ?VA ?PA)
            (not (match-as-part ?NA ?PA))
            (match-as-part ?NA ?)
            (not (match-as-part ?NA ?SS)))
  =>
  (at ?VP (retract ?fact1 ?fact2)
      (assert (view-act world-control-factor ?NA))))
```

状態S.7 で、view-actのmatch-as-part スロットとperforming-actが一致し、next-actのmatch-as-part スロットとperforming-actが一致しない場合、次の動作の比較に移るために、next-actをview-actに代入する。アルゴリズムのStep11, Step13と対応。

R.23 : devide-parts-act-1

```
(defrule devide-parts-act-1 "If PA=map(VA)=map(MA),
            [Case1] WCF has Pinit,
                     whole-act(VA)>parts(PA)=x, whole-act(NA)>parts(PA)=y,
                       then PA <- Pinit, map(VA) <- x, map(NA) <- y"
  (declare (salience -220))
  (viewpoint ?VP
             (schema world-control-factor
                     ?fact1 <- (performing-act ?PA)
                     (view-act ?VA)
                     (next-act ?NA)
                     ?fact2 <- (P-init ?Pinit))
             ?fact3 <- (match-as-part ?VA ?PA)
             ?fact4 <- (match-as-part ?NA ?PA)
             (parts ?PA ?x)
             (or (whole-act ?VA ?x)
                 (match ?VA ?x)
                 (and (match ?VA ?m-va)
                      (or (ako ?m-va ?x)
                          (ako ?x ?m-va))))
             (parts ?PA ?y)
             (or (whole-act ?NA ?y)
                 (match ?NA ?y)
                 (and (match ?NA ?m-na)
                      (or (ako ?m-na ?y)
                          (ako ?y ?m-NA)))))
 =>
  (at ?VP (sprout (retract ?fact1 ?fact2 ?fact3 ?fact4)
                  (assert (performing-act world-control-factor ?Pinit)
                          (match-as-part ?VA ?x)
                          (match-as-part ?NA ?y)
                          (link ?Pinit ?PA)
                          (undevidable-act world-control-factor ?PA)))))
```

状態S.8 で、performing-actを部分動作に分割するとき、先頭動作（ルールR.24で検索され、P-initに格納）をperforming-actに代入し、view-act、next-actのmatch-as-partスロットに、部分動作の中でそれらを部分とするような動作を検索して代入する。

アルゴリズムStep.12 に対応。

R.24 : devide-parts-act-2

```
(defrule devide-parts-act-2 "If PA=map(VA)=map(MA),
           [Case2] not(WCF has Pinit), PA has parts,
                      then Pinit(WCF) <- search"
  (declare (salience -220))
  (viewpoint ?VP
           (schema world-control-factor
                   (performing-act ?PA)
                   (view-act ?VA)
                   (next-act ?NA)
                   (not (P-init ?)))
           (match-as-part ?VA ?PA)
           (match-as-part ?NA ?PA)
           (parts ?PA ?parts)
           (not (and (condition ?parts (?name ?slot ?value))
                     (not (schema ?name
                                  (?slot ?value)))
                     (not (and (ako ?ako-name ?name)
                               (schema ?ako-name
                                       (?slot ?value))))
                     (not (and (ako ?ako-value ?value)
                               (schema ?name
                                       (?slot ?ako-value))))
                     (not (and (ako ?ako-name ?name)
                               (ako ?ako-value ?value)
                               (schema ?ako-name
                                       (?slot ?ako-value))))))
           (exists (condition ?parts ?)))
 =>
 (at ?VP (sprout (assert (P-init world-control-factor ?parts))
                 (dolist (rest (list*$ (get-schema-value ?PA 'parts)))
                   (unless (equal ?parts rest)
                     (assert (REST-ACT ?PA =rest)))))))
```

状態S.6, S.7, S.9 で、view-act、next-actのmatch-as-part スロットが共に、performing-actと一致する場合、performing-actを部分動作に分割する。このとき、部分動作の先頭の動作をP-initに代入し、残りの動作をperforming-actのREST-ACTスロットに代入する。アルゴリズムのStep11, Step12に対応。

R.25 : devide-parts-act-3

```
(defrule devide-parts-act-3 "If PA=map(VA)=map(MA),
            [Case3] not(WCF has Pinit) not(PA has parts),
                      or(x=ako(PA) has parts) (x=kind(PA) has parts),
                        then PA <- x, map(VA) <- x, map(NA) <- x"
  (declare (salience -230))
  (viewpoint ?VP
            (schema world-control-factor
                    ?fact1 <- (performing-act ?PA)
                    (view-act ?VA)
                    (next-act ?NA)
                    (not (P-init ?)))
            ?fact2 <- (match-as-part ?VA ?PA)
            ?fact3 <- (match-as-part ?NA ?PA)
            (not (parts ?PA ?))
            (or (ako ?PA ?x)
                (ako ?x ?PA))
            (parts ?x ?))
  =>
  (at ?VP (sprout (retract ?fact1)
                  (retract ?fact2)
                  (retract ?fact3)
                  (assert (performing-act world-control-factor ?x)
                          (match-as-part ?VA ?x)
                          (match-as-part ?NA ?x)))))
```

状態S.7 で、view-act、next-actのmatch-as-part スロットが共に、performing-actと一致する場合で、performing-actがparts スロットを持たない、即ち、部分動作に分割できないとき、performing-actを中心とした上位ー下位階層の中から、parts スロットを持つ動作を検索し、その動作をperforming-actと、view-act、next-actのmatch-as-part スロットに代入する。この後、ルールR.24で部分動作に分割される。

アルゴリズムStep.11 、Step.12 に対応。

R.26 : poison-world-for-order

```
(defrule poison-world-for-order "If PA=map(NA), PA<>map(VA),
                                        then poison world"
  (viewpoint ?VP (schema world-control-factor
                         (performing-act ?PA)
                         (next-act ?NA)
                         (view-act ?VA))
            (match-as-part ?NA ?PA)
            (match-as-part ?VA ?)
            (not (match-as-part ?VA ?PA)))
  =>
  (at ?VP (poison "restriction of order")))
```

状態S.7 で、next-actのmatch-as-part スロットとperforming-actが一致し、view-act match-as-part スロットとperforming-actが一致しない場合、順序に矛盾があると見なし、この仮説世界を消滅させる。

アルゴリズムのStep11, Step14に対応。

R.27 : poison-world-for-match-count

```
(defrule poison-world-for-match-count "If end, match-factor is less,
                                        poison world"
  (Viewpoint ?VP
            (schema world-control-factor
                    (matching-phase Phase-2)
                    (similar-service ?SS)
                    (performing-act ?SS))
            (status ?SS done)
            (max-match-factor control-factor ?max-factor))
  (Viewpoint ?VP1
            (schema world-control-factor
                    (match-factor ?factor&:(?factor < ?max-factor))
                    (similar-service ?)))
  =>
  (at ?Vp1
      (poison "This world has a less match-factor than the another world")))
```

類似サービスが一つでも見つかった後、類似サービスの持つmatch-factorより、小さなmatch-factorしか持たない仮説世界を消滅させる。

R.28 : Make-result-world

```
(defrule Make-result-world ""
  (declare (salience 100))
  (Viewpoint ?VP
            (schema world-control-factor
                    (similar-service ?SS)
                    ?fact <- (matching-phase Phase-0)))
  (Viewpoint ?VP1
            (schema world-control-factor
                    (matching-phase Phase-6)
                    (Input-service ?Input)
                    (similar-service ?SS))
            (status ?SS done))
  =>
  (at ?VP (sprout (retract ?fact)
                  (assert (matching-phase world-control-factor Phase-3))
                  (dolist (name (list*$ (get-schema-value ?Input 'parts ?VP)))
                    (retract-schema-value name 'match-as-part)
                    (When (get-schema-value name 'match-as-part ?VP1)
                      (put-schema-value name
                                        'match-as-part
                                        (get-schema-value name
                                                          'match-as-part
                                                          ?VP1)))
                    (When (get-schema-value name 'match-intention ?VP1)
                      (put-schema-value name
                                        'match-intention
                                      (get-schema-value name
                                                        'match-intention
                                                        ?VP1))))))
  (at ?root (assert (match-Phase control-factor find))))
```

デバッグ用ルール。下線部のフェーズまで推論が進んだ時点で、類似サービスを仮定した仮説世界より、新しい仮説世界が生成され、その仮説世界に推論結果が代入される。

R.29 : find-similar-service

```
(defrule find-similar-service ""
  (declare (salience -5000))
  (Viewpoint ?VP1
            (schema world-control-factor
                    ?fact1 <- (matching-phase Phase-2)
                    (Input-service ?Input)
                    (similar-service ?SS)
                    ?fact2 <- (view-act ?VA))
            (status ?SS done)
            (schema control-factor
                    (max-match-factor ?MMF)
                    (act-number ?AN)))
 =>
 (at ?root (assert (schema control-factor
                            (match-Phase find)
                            (max-match-intention-factor =(- ?AN ?MMF)))))
 (at ?VP1
     (retract ?fact1 ?fact2)
     (assert (schema world-control-factor
                     (matching-phase Phase-4)
                     (match-intention-factor =(- ?AN ?MMF))))))
```

最初の類似サービスを発見したときに発火するルール。次のフェーズへ渡す情報を作成すると共に、このルール発火後は、max-match-factorより小さいmatch-factorしか持たない世界での推論が停止する。

R.30 : poison-unrelated-world-prep

```
(defrule poison-unrelated-world-prep ""
  (declare (salience -300000))
  =>
  (assert (search-similar-service-Phase-end)))
```

デバッグ用ルール。ルールR.31が発火するために、フェーズを変えている。

R.31 : poison-unrelated-world

```
(defrule poison-unrelated-world ""
  (declare (salience -300000))
  (Viewpoint ?VP
            (search-similar-service-Phase-end)
            (matching-phase world-control-factor Phase-1))
 =>
 (at ?VP (poison "poison unrelated world")))
```

デバッグ用ルール。正しい推論と関係のない仮説世界を消滅させる。

```
(undefrule poison-unrelated-world)
```

ルールR.30が発火しないようにするための関数。

4.2. アルゴリズムとルールの対応

以下にアルゴリズムの概要の各Stepと対応するルール、状態、条件分岐を示す。ここで()は省略の可能性を示し、()* は0回以上の繰り返しを示す。また、| は分岐を、・は異なるルールが別々に発火することを、==> は同一ルールが複数回発火することを示す。

```
[前処理] initial --> R.1 --> S.1 (--> C.1 --> R.2 --> S.1)*
[Step1 ] S.1 --> C.1 ==> R.3 --> S.2
[Step2 ] S.2 (--> C.2 --> R.4 --> C.3 --> S.2)*
             (--> C.2 --> R.4 --> C.3 --> S.2 inactive)
[Step3 ] S.2 --> C.2 --> R.6 --> S.3
[Step4 ] S.3 --> R.7 --> S.4
[Step5 ] S.4 --> C.5 --> R.12 --> S.4
[Step6 ]  ・S.4 --> C.5 --> R.11 --> S.4inactive
          ・S.4 --> C.5 --> R.8 --> S.5
[Step7 ] S.5 --> C.6  | --> R.9 --> S.6
                      | --> R.10 --> S.11
[Step8 ] S.6 --> C.7  | --> R.12 --> S.6
                      | --> R.13 --> S.11
[Step9 ]  ・S.6 --> C.7 --> R.11 --> S.6inactive
          ・S.6 --> C.7 --> C.8  | --> Step10
                                 | --> Step12
[Step10] Step9 --> C.8  | ==> R.14 --> S.7
                        | --> dead.state
[Step11] S.7 --> C.10 | --> Step12
                      | --> Step13
                      | --> Step14
                      | --> Step15
[Step12]                              Step9 |
         Step11 --> C.11  | (==> R.25 --> S.9) | ==> R.24 --> S.8==> R.23 --> S.7
                          | --> dead.state
[Step13] Step11 --> R.22 --> S.5
[Step14] Step11 --> R.26 --> poison
[Step15] Step11 --> R.15 --> S.10 --> C.12  | --> dead.state
                                            | ==>  | R.17 --> S.6
                                                   | R.18 --> S.6
                                                   | R.19 --> S.6
                                                   | R.20 --> S.6
                                                   | R.21 --> S.6
[Step16] S11 --> R.16 --> S.12 --> C.13 | --> dead.state
                                        | --> end
                                        | ==>  | R.17 --> S.11
                                               | R.18 --> S.11
                                               | R.19 --> S.11
                                               | R.20 --> S.11
                                               | R.21 --> S.11
```

5. 実行例

付録2に知識ベースの例を、付録3に入力例を示す。付録3には2種類の入力例があるが、この節では、2つめの入力例に対する実行例を示す。実行結果は、図2に示すようなVIEW-POINT構造であり、各VIEW-POINTには図3に示すようなルールによって推論された結果が格納されている。

以下に、各VIEW-POINTにおける推論結果の導出過程について説明する。

(S-1) R.1,R.2 により、Act1のmatch スロットの値lift-handsetを部分に持つような動作を検索し、Act1、及び、world-control-factorのwhole-act スロットに格納。
Act2,Act3,Act4についても同様に行う。

(S-2) R.3 により類似サービスplace-callを仮定。SSに格納。
R.4 によりplace-callがAct1,Act2,Act3,Act4 を部分として含む可能性(即ち、それぞれの動作の whole-actスロットにplace-callが入っているか)を調べ、可能性があればそれぞれの動作の match-as-partに格納。同時に、可能性のある動作の数を調べ(この場合は4)、MFに格納。

cf. 類似サービスが originate-call と仮定した場合は(S-3) (図4参照)

(S-4) R.6 により、place-callが起こりえる初期状態を設定、PAにplace-callを格納。
R.7 により、NAに、入力動作の最初の動作 Act1 を格納。

(S-6) R.8 により、NAの値 Act1 をVAに格納。
R.9 により、Act1の次の動作 Act2 をNAに格納。

(S-7) R.11により、NAの値 Act2 がよくないという仮説をたてる。NAに次の動作 Act3 を格納し、MFの値を1減らし3にする。

cf. (S-4) の時点でこの仮説をたてた場合は(S-5) へ進んでいる。
(S-6) の時点でこの仮説をたてない場合は(S-8) へ進んでいる。

(S-50)R.24により、PAを分解し、先頭の動作 (originate-call) を仮定、p-initへ格納。
その他の動作はPAのrest-actスロットに格納する。 ── ①

(S-51)R.23により、PAにp-initの値(= originate-call) を格納。また、VA(= Act1) と NA(= Act3) のMAP スロットの値が、originate-callであると仮定する。── ②

(S-52)①と同様。p-initはlift-handset-idle になる。

cf. もし、先頭の動作の候補が複数ある場合はp-initの値が違うような世界が複数できる。この場合の例は、(S-72)がある。(図4参照)

(S-53)②と同様。PAはlift-handset-idle 、Act1のMAP スロットはlift-handset-idle 、Act3のMAP スロットはclose-loopになる。
更に、R.22により、VAの値はAct1からAct3になり、R.9 によりNAの値はAct3の次の動作Act4になる。

(S-55)R.14により、NA(= Act4) のMAP スロットの値が SS(= place-call)であるので、MAP スロットの値を、talk-telephoneであると仮定し、変える。
また、R.15により、perform-act を実行する。(状態が変化する。)

(S-56)R.17により、perform-act をlift-handset-idle からpercieve-lift-handset に変える。また、R.15により、perform-act を実行する。(状態が変化する。)

これ以後の推論は省略する。このようにして、最終的に図2に示すようなview-point構造が得られる。

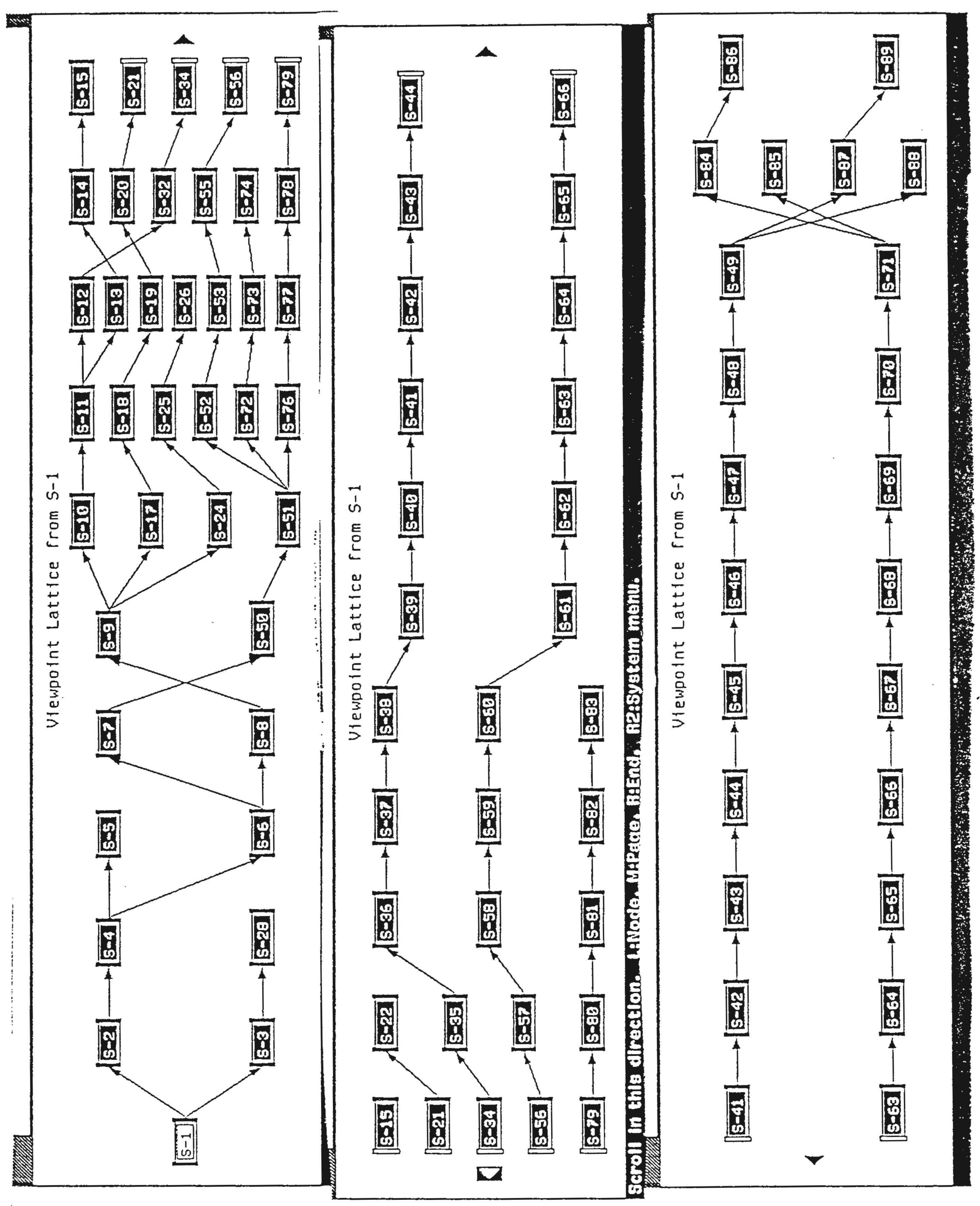

図２．結果のVIEW-POINT構造

図3. 各VIEW-POINT内の事実1

```
S-1
f-2489 [SEARCH-SIMILAR-SERVICE-PHASE-END]
f-2466 [MAX-MATCH-INTENTION-FACTOR CONTROL-FACTOR 1]
f-2465 [MATCH-PHASE CONTROL-FACTOR FIND]
f-2220 [MAX-MATCH-FACTOR CONTROL-FACTOR 3]
f-2080 [WHOLE-ACT ACT4 PLACE-CALL]
f-2079 [WHOLE-ACT ACT3 PLACE-CALL]
f-2078 [WHOLE-ACT ACT3 ORIGINATE-CALL]
f-2077 [WHOLE-ACT ACT2 PLACE-CALL]
f-2076 [WHOLE-ACT ACT2 ORIGINATE-CALL]
f-2075 [WHOLE-ACT ACT1 PLACE-CALL]
f-2074 [WHOLE-ACT WORLD-CONTROL-FACTOR PLACE-CALL]
f-2073 [WHOLE-ACT WORLD-CONTROL-FACTOR ORIGINATE-CALL]
f-2072 [WHOLE-ACT ACT1 ORIGINATE-CALL]
f-2071 [MATCHING-PHASE WORLD-CONTROL-FACTOR SEARCH-WHOLE-ACTION] in
```

```
S-2
-----------------------
f-2102 [MATCH-FACTOR WORLD-CONTROL-FACTOR 4] in [S-2|S-12, S-7, S-5
f-2101 [MATCH-AS-PART ACT4 PLACE-CALL] in [S-2|S-55, S-32)
f-2099 [MATCH-AS-PART ACT3 PLACE-CALL] in [S-2|S-51, S-13, S-12)
f-2097 [MATCH-AS-PART ACT2 PLACE-CALL] in [S-2|S-9, S-7)
f-2095 [MATCH-AS-PART ACT1 PLACE-CALL] in [S-2|S-51, S-9, S-5)
f-2084 [MATCHING-PHASE WORLD-CONTROL-FACTOR PHASE-0] in [S-2|S-4)
f-2083 [PARTIAL-ACTION WORLD-CONTROL-FACTOR PLACE-CALL] in [S-2)
f-2081 [SIMILAR-SERVICE WORLD-CONTROL-FACTOR PLACE-CALL] in [S-2)
```

```
S-4
-----------------------
f-2112 [NEXT-ACT WORLD-CONTROL-FACTOR ACT1] in [S-4|S-5, S-6)
f-2111 [PERFORMING-ACT WORLD-CONTROL-FACTOR PLACE-CALL] in [S-4|S-51
f-2110 [MATCHING-PHASE WORLD-CONTROL-FACTOR PHASE-1] in [S-4|S-58, S
f-2109 [HAS-A RECEIVER-HUMAN RECEIVER-TELEPHONE] in [S-4)
f-2108 [HAS-A CALLER-HUMAN CALLER-TELEPHONE] in [S-4)
f-2107 [STATE RECEIVER-TELEPHONE TELEPHONE-ONHOOK] in [S-4|S-71, S-4
f-2106 [STATE RECEIVER-HUMAN HUMAN-IDLE] in [S-4|S-68, S-46)
f-2105 [STATE CO-EXCHANGE CO-EXCHANGE-IDLE] in [S-4|S-66, S-44)
f-2104 [STATE CALLER-TELEPHONE TELEPHONE-ONHOOK] in [S-4|S-66, S-44
f-2103 [STATE CALLER-HUMAN HUMAN-IDLE] in [S-4|S-55, S-32, S-13)
```

```
S-6
-----------------------
f-2115 [NEXT-ACT WORLD-CONTROL-FACTOR ACT2] in [S-6|S-7, S-11)
f-2114 [VIEW-ACT WORLD-CONTROL-FACTOR ACT1] in [S-6|S-53, S-11)
```

```
S-7
-----------------------
f-2229 [NEXT-ACT WORLD-CONTROL-FACTOR ACT3] in [S-7|S-53)
f-2116 [MATCH-FACTOR WORLD-CONTROL-FACTOR 3] in [S-7)
```

```
S-50
-----------------------
f-2307 [REST-ACT PLACE-CALL TALK-TELEPHONE] in [S-50|S-70)
f-2306 [REST-ACT PLACE-CALL ANSWER-CALL] in [S-50|S-69)
f-2305 [REST-ACT PLACE-CALL ALERT] in [S-50|S-68)
f-2304 [REST-ACT PLACE-CALL TRANSMIT-TEL-NO] in [S-50|S-67)
f-2303 [P-INIT WORLD-CONTROL-FACTOR ORIGINATE-CALL] in [S-50|S-51)
```

```
S-51
-----------------------
f-2312 [UNDEVIDABLE-ACT WORLD-CONTROL-FACTOR PLACE-CALL] in [S-51)
f-2311 [LINK ORIGINATE-CALL PLACE-CALL] in [S-51|S-67)
f-2310 [MATCH-AS-PART ACT3 ORIGINATE-CALL] in [S-51|S-77, S-73, S-53
f-2309 [MATCH-AS-PART ACT1 ORIGINATE-CALL] in [S-51|S-77, S-73, S-53
f-2308 [PERFORMING-ACT WORLD-CONTROL-FACTOR ORIGINATE-CALL] in [S-51
```

```
S-52
-----------------------
f-2323 [REST-ACT ORIGINATE-CALL LISTEN-DT] in [S-52|S-65)
f-2322 [REST-ACT ORIGINATE-CALL SEND-VOICE] in [S-52|S-64)
f-2321 [REST-ACT ORIGINATE-CALL VOICE-SIGNAL-TO-VOICE] in [S-52|S-6
f-2320 [REST-ACT ORIGINATE-CALL RECEIVE-VOICE-SIGNAL] in [S-52|S-62
f-2319 [REST-ACT ORIGINATE-CALL SEND-DT-SIGNAL] in [S-52|S-61)
f-2318 [REST-ACT ORIGINATE-CALL CLOSE-LOOP-TO-SEND-DT] in [S-52|S-6
f-2317 [REST-ACT ORIGINATE-CALL PERCEIVE-CLOSE-LOOP-IDLE] in [S-52|
f-2316 [REST-ACT ORIGINATE-CALL CLOSE-LOOP] in [S-52|S-58)
f-2315 [REST-ACT ORIGINATE-CALL HANDSET-UP-TO-LOOP-CLOSE] in [S-52|
f-2314 [REST-ACT ORIGINATE-CALL PERCEIVE-LIFT-HANDSET] in [S-52|S-5
f-2313 [P-INIT WORLD-CONTROL-FACTOR LIFT-HANDSET-IDLE] in [S-52|S-5
```

```
S-53
-----------------------
f-2331 [NEXT-ACT WORLD-CONTROL-FACTOR ACT4] in [S-53|S-58)
f-2330 [VIEW-ACT WORLD-CONTROL-FACTOR ACT3] in [S-53|S-58)
f-2329 [INIT-ACT WORLD-CONTROL-FACTOR LIFT-HANDSET-IDLE] in [S-53)
f-2328 [UNDEVIDABLE-ACT WORLD-CONTROL-FACTOR ORIGINATE-CALL] in [S-5
f-2327 [LINK LIFT-HANDSET-IDLE ORIGINATE-CALL] in [S-53|S-56)
f-2326 [MATCH-AS-PART ACT3 CLOSE-LOOP] in [S-53)
f-2325 [MATCH-AS-PART ACT1 LIFT-HANDSET-IDLE] in [S-53)
f-2324 [PERFORMING-ACT WORLD-CONTROL-FACTOR LIFT-HANDSET-IDLE] in [S
```

```
S-55
-----------------------
f-2335 [STATUS LIFT-HANDSET-IDLE DONE] in [S-55|S-56)
f-2334 [STATE HUMAN HUMAN-HANDSET-UP] in [S-55)
f-2333 [MATCH-AS-PART ACT4 TALK-TELEPHONE] in [S-55)
```

```
S-56
-----------------------
f-2339 [STATUS PERCEIVE-LIFT-HANDSET DONE] in [S-56|S-57)
f-2338 [STATE TELEPHONE TELEPHONE-HANDSET-UP] in [S-56)
f-2337 [LINK PERCEIVE-LIFT-HANDSET ORIGINATE-CALL] in [S-56|S-57)
f-2336 [PERFORMING-ACT WORLD-CONTROL-FACTOR PERCEIVE-LIFT-HANDSET] i
```

```
S-3
------------------------
f-2094 [MATCH-FACTOR WORLD-CONTROL-FACTOR 3] in [S-3)
f-2093 [MATCH-AS-PART ACT3 ORIGINATE-CALL] in [S-3)
f-2091 [MATCH-AS-PART ACT2 ORIGINATE-CALL] in [S-3)
f-2089 [MATCH-AS-PART ACT1 ORIGINATE-CALL] in [S-3)
f-2088 [MATCHING-PHASE WORLD-CONTROL-FACTOR PHASE-0] in [S-3|S-28)
f-2087 [PARTIAL-ACTION WORLD-CONTROL-FACTOR ORIGINATE-CALL] in [S-3
f-2085 [SIMILAR-SERVICE WORLD-CONTROL-FACTOR ORIGINATE-CALL] in [S-3

S-28
------------------------
f-2234 [NEXT-ACT WORLD-CONTROL-FACTOR ACT1] in [S-28)
f-2226 [PERFORMING-ACT WORLD-CONTROL-FACTOR ORIGINATE-CALL] in [S-28
f-2225 [MATCHING-PHASE WORLD-CONTROL-FACTOR PHASE-1] in [S-28)
f-2224 [HAS-A CALLER-HUMAN CALLER-TELEPHONE] in [S-28)
f-2223 [STATE CO-EXCHANGE CO-EXCHANGE-IDLE] in [S-28)
f-2222 [STATE CALLER-TELEPHONE TELEPHONE-ONHOOK] in [S-28)
f-2221 [STATE CALLER-HUMAN HUMAN-IDLE] in [S-28)

S-72
f-2407 [REST-ACT ORIGINATE-CALL SEND-DT-SIGNAL] in [S-72)
f-2406 [REST-ACT ORIGINATE-CALL CLOSE-LOOP-TO-SEND-DT] in [S-72)
f-2405 [REST-ACT ORIGINATE-CALL PERCEIVE-CLOSE-LOOP-IDLE] in [S-72)
f-2404 [REST-ACT ORIGINATE-CALL CLOSE-LOOP] in [S-72)
f-2403 [REST-ACT ORIGINATE-CALL HANDSET-UP-TO-LOOP-CLOSE] in [S-72|S
f-2402 [REST-ACT ORIGINATE-CALL LIFT-HANDSET-IDLE] in [S-72)
f-2401 [P-INIT WORLD-CONTROL-FACTOR PERCEIVE-LIFT-HANDSET] in [S-72|

S-73
------------------------
f-2419 [STATUS PERCEIVE-LIFT-HANDSET DONE] in [S-73|S-74)
f-2418 [STATE TELEPHONE TELEPHONE-HANDSET-UP] in [S-73)
f-2417 [INIT-ACT WORLD-CONTROL-FACTOR PERCEIVE-LIFT-HANDSET] in [S-7
f-2416 [UNDEVIDABLE-ACT WORLD-CONTROL-FACTOR ORIGINATE-CALL] in [S-7
f-2415 [LINK PERCEIVE-LIFT-HANDSET ORIGINATE-CALL] in [S-73|S-74)
f-2414 [MATCH-AS-PART ACT3 CLOSE-LOOP] in [S-73)
f-2413 [MATCH-AS-PART ACT1 LIFT-HANDSET-IDLE] in [S-73)
f-2412 [PERFORMING-ACT WORLD-CONTROL-FACTOR PERCEIVE-LIFT-HANDSET] i

S-74
------------------------
f-2422 [STATUS HANDSET-UP-TO-LOOP-CLOSE DONE] in [S-74)
f-2421 [LINK HANDSET-UP-TO-LOOP-CLOSE ORIGINATE-CALL] in [S-74)
f-2420 [PERFORMING-ACT WORLD-CONTROL-FACTOR HANDSET-UP-TO-LOOP-CLOSE

Viewpoint Lattice from S-66
S-5  S-7  S-9  S-10
S-17
S-24
S-6  S-8  S-50  S-51

COMMAND WINDOW
=>
=> text
=> help
=> help
=> help
=> help
=> S-5
=> facts

FACTS
f-2227 [NEXT-ACT WORLD-CONTROL-FACTOR ACT2] in [S-5)
f-2113 [MATCH-FACTOR WORLD-CONTROL-FACTOR 3] in [S-5)

L: f-2113 [MATCH-FACTOR WORLD-CONTROL-FACTOR 3] in [S-5) L2: Window menu. M: Pop. M2: Root. R: Help for this option.
[Wed 7 Feb 5:12:38]   al   Keyboard        CL USER:        User Input
```

図4. 各VIEW-POINT内の事実2

6．今後の展開

本稿では、類似サービス検索部の前半部の実現手法について述べた。今後は、入力動作フレームと類似サービス中の動作の類似を用いたサービスの検索部分を実現しなければならない。

前半部での出力であるVIEW-POINT構造は、類似サービス中の動作が実行された時点と、入力動作が類似サービス中の動作が一致した時点に対し、それぞれVIEW-POINTを持っている。このVIEW-POINT構造を利用した後半部のアルゴリズムを以下に示す。

[STEP1] 類似サービス中の最初の動作(init-act)と最後の動作(last-act)を検索する。

[STEP2] 一致しなかった入力動作（即ち、MAP スロットに値を持たない入力動作）の数をMIF(match-intention-factor) に入れる。

[STEP3] 一致しなかった入力動作の中で、最も最初に現れる動作をＡとする。

[STEP4] Ａの前の動作(pre-actスロットの値) をＢとする。もし、Ａがpre-act スロットに値を持たなければ、init-actをＣとして、STEP6 へ行く。

[STEP5] ＢのMAP またはsimilar-act スロットの値をＣとする。もし、Ｂがこれらのスロットに値を持たなければ、Ｂのpre-act の値をＢとしてSTEP5 へ行く。ここで、Ｂがpre-act に値を持たなければ、init-actをＣとする。

[STEP6] Ａの後の動作(post-act スロットの値) をＤとする。もし、Ａがpost-actスロットに値を持たなければ、last-actをＥとして、STEP8 へ行く。

[STEP7] ＤのMAP スロットの値をＥとする。もし、ＤがMAP スロットに値を持たなければ、Ｄのpost-actをＤとしてSTEP7 へ行く。ここで、Ｂがpost-actに値を持たなければ、init-actをＣとする。

[STEP8] Ｃが実行されたVIEW-POINTの次のVIEW-POINTから、Ｅが実行されたVIEW-POINTの前のVIEW-POINTまでに実行された動作の中から、Ａの類似動作を検索する。類似動作の判定は、message-from,message-to,intention スロットの値によって行う。類似動作が見つかれば、その動作をＡのsimilar-act スロットに入れて、STEP10へ行く。類似動作が見つからない場合はSTEP9 へ行く。

[STEP9] MIF の値を１減らす。

[STEP10] 次の一致しなかった入力動作があればその動作をＡとしてSTEP4 へ、無ければ終了する。

後半部分のプログラムは実現中であり、未完成であるが、付録４に参考として付しておく。

参考文献

［１］　横田、島：「通信ソフトウェアにおける要求理解の一手法」、情報処理学会 知識工学と人工知能　研究会報告 No.68、１９９０．

［２］　Bruce D. Clayton : "A First Look at Viewpoints", ＡＲＴ Programing Tutorial Volume Two.

```
;;; -*- Mode: ART; Base: 10.; Package: ART-USER; Syntax: COMMON-LISP -*-

(def-viewpoint-levels (s merging nil))

(defschema part-of "inh-rel"
  (inverse parts)
  (instance-of relation))

(defschema parts "rel"
  (inverse part-of)
  (instance-of relation))

(defschema ako "inh-rel"
  (inverse kind)
  (instance-of inh-relation)
  (element-of inh-relations)
  (transitivity (repeat (step ako $) 1 inf)))

(defschema kind "rel"
  (inverse ako)
  (instance-of relation))

;;(defschema one-parts "one level relation for parts"
;;  (inverse one-part-of)
;;  (instance-of relation))
;;
;;(defschema one-part-of "one level relation for part-of"
;;  (inverse one-parts)
;;  (instance-of relation))
;;
;;(defschema part-of "inh-rel for part only"
;;  (inverse parts)
;;  (instance-of inh-relation)
;;  (transitivity (repeat (or (sequence (step ako $)
;;                                      (step one-part-of $))
;;                            (sequence (step kind $)
;;                                      (step one-part-of $))
;;                            (step one-part-of $)) 1 inf)))
;;
;;(defschema parts "inh-rel for part only"
;;  (inverse part-of)
;;  (instance-of relation))

(defschema match "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema match-as-part "rel"
  (instance-of slot))

(defschema match-intention "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema whole-act "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema PRE-ACT "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema POST-ACT "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema REST-ACT "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema undevidable-act "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema STATUS "rel"
  (instance-of slot)
  (slot-how-many single-value))
```

```
(defschema LINK "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema CONDITION "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema Change-from "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema Change-To "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema OUTPUT-ACT "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema INPUT-ACT "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema INVERSE-ACT "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema state "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema has-a "rel"
  (instance-of slot)
  (slot-how-many multiple-values))

(defschema agent "rel"
  (instance-of slot)
    (slot-how-many single-value))

(defschema message-from "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema message-to "rel"
  (instance-of slot)
  (slot-how-many single-value))

(defschema control-factor "factor of most adaptive world"
  (act-number)
  (max-match-factor)
  (max-match-intention-factor)
  (match-phase))

(defschema world-control-factor "factor for matching in the world"
  (match-factor)
  (match-intention-factor)
  (whole-act)
  (similar-service)
  (PARTIAL-ACTION)
  (PERFORMING-ACT)
  (VIEW-ACT)
  (NEXT-ACT)
  (P-init)
  (Input-service)
  (pre-act)
  (init-act)
  (last-act)
  (matching-phase)
  (match-intention-act)
  (undevidable-act))
```

```
(defrule search-whole-action-mode "control-fact-for-initial"
  (declare (salience 1000))
  (viewpoint ?VP
            (schema control-factor
                    (act-number ?AN)
                    (not (max-match-factor ?))))
  =>
  (at ?VP (assert (max-match-factor control-factor ?AN)
                  (matching-phase world-control-factor search-whole-action))))

(defrule search-whole "search-whole-action"
  (declare (salience 100))
  (viewpoint ?VP
            (matching-phase world-control-factor search-whole-action)
            (or (match ?inp ?act)
                (whole-act ?inp ?act))
            (or (part-of ?act ?whole-act)
                (and (or (ako ?act ?act1)
                         (ako ?act1 ?act))
                     (part-of ?act1 ?whole-act))))
  =>
  (at ?VP (assert (whole-act ?inp ?whole-act)
                  (whole-act world-control-factor ?whole-act))))

(DEFRULE similar-service-hypotheses "make hypotheses"
  (declare (salience -100))
  (viewpoint ?VP
           (schema world-control-factor
                   ?fact <- (matching-phase search-whole-action)
                   (whole-act ?act)))
  =>
  (at ?VP (sprout (retract ?fact)
                  (assert (schema world-control-factor
                                  (similar-service ?act)
                                  (match-factor 0)
                                  (partial-action ?act)
                                  (matching-phase phase-0))))))

(defrule set-match-as-part "set match-as-part slot of act"
  (declare (salience -110))
  (viewpoint ?vp
           (schema world-control-factor
                   (matching-phase phase-0)
                   (similar-service ?act)
                   ?fact <- (match-factor ?factor))
           (schema ?input-act
                   (match ?)
                   (whole-act ?act))
           (not (match-as-part ?input-act ?)))
  =>
  (at ?vp
      (retract ?fact)
      (assert (match-as-part ?input-act ?act)
              (match-factor world-control-factor =(+ ?factor 1)))))

(defrule change-control-factor ""
  (declare (salience -200000))
  (viewpoint ?VP
           ?fact <- (max-match-factor control-factor ?max-factor)
           (not (match-phase control-factor ?)))
  =>
  (at ?VP
      (retract ?fact)
      (assert (max-match-factor control-factor =(- ?max-factor 1)))))
```

```
(defrule set-initial-state "set initial state"
  (declare (salience -120))
  (viewpoint ?VP
             (schema world-control-factor
                     ?fact <- (matching-phase phase-0)
                     (similar-service ?act)
                     (match-factor ?factor))
             (max-match-factor control-factor ?max-factor&:(?max-factor = ?factor)))
  =>
  (at ?VP (sprout (retract ?fact)
                  (dolist (cond (list*$ (get-schema-value ?Act 'condition)))
                    (put-schema-value (first (list*$ cond))
                                      (second (list*$ cond))
                                      (third (list*$ cond))))
                  (assert (schema world-control-factor
                                  (matching-phase Phase-1)
                                  (performing-act ?act))))))

(defrule set-initial-next-act "If VA=nil, NA=nil, set NA"
  (declare (salience -200))
  (viewpoint ?VP
             (schema world-control-factor
                     (matching-phase Phase-1)
                     (not (view-act ?))
                     (not (next-act ?)))
             (match ?act ?)
             (not (pre-act ?act ?)))
  =>
  (at ?VP (assert (next-act world-control-factor ?act))))

(defrule set-initial-view-act "If VA=nil, NA=t, map(NA)=similar-service, VA <- NA, NA<- nil"
  (declare (salience -200))
  (viewpoint ?VP
             (schema world-control-factor
                     (not (view-act ?))
                     ?fact <- (next-act ?NA)
                     (similar-service ?map))
             (match-as-part ?NA ?map))
  =>
  (at ?VP (sprout
           (retract ?fact)
           (assert (view-act world-control-factor ?NA)))))

(defrule set-next-act "If VA=t, NA=nil, NA <- post(VA)"
  (declare (salience -200))
  (viewpoint ?VP
             (schema world-control-factor
                     (matching-phase Phase-1)
                     (match-factor ?factor)
                     (view-act ?VA)
                     (not (next-act ?)))
             (max-match-factor control-factor ?max-factor&:(?max-factor = ?factor))
             (post-act ?VA ?post))
  =>
  (at ?VP (assert (next-act world-control-factor ?post))))

(defrule cant-set-next-act "If VA=t, NA=nil, post(VA)=nil, change mode"
  (declare (salience -200))
  (viewpoint ?VP
             (schema world-control-factor
                     ?fact <- (matching-phase Phase-1)
                     (view-act ?VA)
                     (not (next-act ?)))
             (not (post-act ?VA ?)))
  =>
  (at ?VP
      (retract ?fact)
      (assert (matching-phase world-control-factor Phase-2))))
```

```
(defrule hyp-next-act-inadequate "If NA=similar-service, map(NA)=t, then hyp map(NA)=nil"
  (declare (salience -190))
  (viewpoint ?VP
             (schema world-control-factor
                     (next-act ?NA)
                     (similar-service ?SS)
                     ?fact1 <- (match-factor ?factor))
             (max-match-factor control-factor ?max-factor&:(?max-factor = ?factor))
             ?fact2 <- (match-as-part ?NA ?SS))
  =>
  (at ?VP (sprout
            (retract ?fact1 ?fact2)
            (assert (match-factor world-control-factor =(- ?factor 1))))))

(defrule retract-inadequate-next-act
         "If map(NA)=nil, NA <- post(NA)"
  (declare (salience -190))
  (viewpoint ?VP
             (max-match-factor control-factor ?max-factor)
             (match-factor world-control-factor ?factor&:(?factor = ?max-factor))
             ?fact <- (next-act world-control-factor ?NA)
             (not (match-as-part ?NA ?))
             (post-act ?NA ?PA))
;;  (similar-service world-control-factor ?)
  =>
  (at ?VP
      (retract ?fact)
      (assert (next-act world-control-factor ?PA))))

(defrule retract-inadequate-next-act-2
         "If map(NA)=nil, post(NA)=nil, Change Mode"
  (declare (salience -200))
  (viewpoint ?VP
             (max-match-factor control-factor ?max-factor)
             (schema world-control-factor
                     (match-factor ?factor&:(?factor = ?max-factor))
                     ?fact <- (matching-phase Phase-1)
                     ?fact1 <- (next-act ?NA))
             (not (match-as-part ?NA ?))
             (not (post-act ?NA ?)))
  =>
  (at ?VP
      (retract ?fact ?fact1)
      (assert (matching-phase world-control-factor Phase-2))))

(defrule change-map-of-next-act "If map(NA)=similar-service, whole(NA) == (rest or PA),
                                 then (sprout), map(NA) <- (rest or PA)"
  (declare (salience -200))
  (viewpoint ?VP
             (schema world-control-factor
                     (next-act ?NA)
                     (similar-service ?map))
             ?fact <- (match-as-part ?NA ?map)
             (or (whole-act ?NA ?whole&~?map)
                 (match ?NA ?whole)
                 (and (match ?NA ?x)
                      (or (ako ?x ?whole)
                          (ako ?whole ?x))))
             (or (performing-act world-control-factor ?whole)
                 (REST-ACT ? ?whole)))
  =>
  (at ?VP (sprout (retract ?fact)
           (assert (match-as-part ?NA ?whole)))))
```

```
(defrule perform-act "If PA <> map(VA), PA<> map(NA), then perform PA, status(PA) <- done"
  (declare (salience -210))
  (viewpoint ?VP
             (schema world-control-factor
                     (performing-act ?PA)
                     (view-act ?VA)
                     (next-act ?NA)
                     (similar-service ?SS))
             (not (status ?PA done))
             (not (match-as-part ?VA ?PA))
             (not (match-as-part ?NA ?PA))
             (not (match-as-part ?NA ?SS))
             (match-as-part ?NA ?))
  =>
  (at ?VP
      (dolist (from (list*$ (get-schema-value ?PA 'change-from)))
        (retract-schema-value (first (list*$ from))
                              (second (list*$ from)) nil (third (list*$ from))))
      (dolist (to (list*$ (get-schema-value ?PA 'change-to)))
        (put-schema-value (first (list*$ to)) (second (list*$ to)) (third (list*$ to))))
      (assert (status ?PA done))))

(defrule perform-act-after-match ""
  (declare (salience -210))
  (viewpoint ?VP
             (schema world-control-factor
                     (matching-phase Phase-2)
                     (performing-act ?PA)
                     (view-act ?)
                     (not (next-act ?))))
  =>
  (at ?VP
      (dolist (from (list*$ (get-schema-value ?PA 'change-from)))
        (retract-schema-value (first (list*$ from))
                              (second (list*$ from)) nil (third (list*$ from))))
      (dolist (to (list*$ (get-schema-value ?PA 'change-to)))
        (put-schema-value (first (list*$ to)) (second (list*$ to)) (third (list*$ to))))
      (assert (status ?PA done))))

(defrule next-perform-act-1
         "If status(PA)=done, ako(PA)=composite-act, then search PA"
  (declare (salience -210))
  (viewpoint ?VP
             ?fact <- (performing-act world-control-factor ?PA)
             ?fact1 <- (status ?PA done)
             ?fact2 <- (link ?PA ?whole)
             ?fact3 <- (rest-act ?whole ?rest)
             (ako ?PA composite-act)
             (not (and (condition ?rest (?name ?slot ?value))
                       (not (schema ?name
                                    (?slot ?value)))
                       (not (and (ako ?ako-name ?name)
                                 (schema ?ako-name
                                         (?slot ?value))))
                       (not (and (ako ?ako-value ?value)
                                 (schema ?name
                                         (?slot ?ako-value))))
                       (not (and (ako ?ako-name ?name)
                                 (ako ?ako-value ?value)
                                 (schema ?ako-name
                                         (?slot ?ako-value))))))
  =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))
```

```
(defrule next-perform-act-2
        "If status(PA)=done, ako(PA)=function, then PA <- output-Act(PA)"
  (declare (salience -210))
  (viewpoint ?VP
             ?fact <- (performing-act world-control-factor ?PA)
             ?fact1 <- (status ?PA done)
             ?fact2 <- (link ?PA ?whole)
             ?fact3 <- (rest-act ?whole ?rest)
             (ako ?PA functional-act)
             (or (output-act ?PA ?rest)
                 (and (output-act ?PA ?x)
                      (or (ako ?x ?rest)
                          (ako ?rest ?x)))))
  =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))

(defrule next-perform-act-3
        "If status(PA)=done, ako(PA)=send-message, then PA <- inverse-act(PA)"
  (declare (salience -210))
  (viewpoint ?VP
             ?fact <- (performing-act world-control-factor ?PA)
             ?fact1 <- (status ?PA done)
             ?fact2 <- (link ?PA ?whole)
             ?fact3 <- (rest-act ?whole ?rest)
             (ako ?PA send-message)
             (or (inverse-act ?PA ?rest)
                 (and (inverse-act ?PA ?x)
                      (or (ako ?x ?rest)
                          (ako ?rest ?x)))))
  =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))

(defrule next-perform-act-4
        "If status(PA)=done, ako(PA)=receive-massage, input(X)=PA, then PA <- X "
  (declare (salience -210))
  (viewpoint ?VP
             ?fact <- (performing-act world-control-factor ?PA)
             ?fact1 <- (status ?PA done)
             ?fact2 <- (link ?PA ?whole)
             ?fact3 <- (rest-act ?whole ?rest)
             (ako ?PA receive-message)
             (or (input-act ?rest ?PA)
                 (and (input-act ?x ?PA)
                      (or (ako ?x ?rest)
                          (ako ?rest ?x)))))
  =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2 ?fact3)
                  (assert (performing-act world-control-factor ?rest)
                          (link ?rest ?whole)))))

(defrule next-perform-act-5
        "If status(PA)=done, rest(whole-act(PA)):nil, then PA <- whole-act(PA)"
  (declare (salience -210))
  (viewpoint ?VP
             ?fact <- (performing-act world-control-factor ?PA)
             ?fact1 <- (status ?PA done)
             ?fact2 <- (link ?PA ?whole)
             (not (rest-act ?whole ?)))
  =>
  (at ?VP (sprout (retract ?fact ?fact1 ?fact2)
                  (assert (performing-act world-control-factor ?whole)))))
```

```
(defrule next-action "If PA=map(VA), PA<>map(NA), then VA <- NA, NA <- nil"
  (declare (salience -210))
  (viewpoint ?VP
            (schema world-control-factor
                    (performing-act ?PA)
                    ?fact1 <- (view-act ?VA)
                    ?fact2 <- (next-act ?NA)
                    (similar-service ?SS))
            (match-as-part ?VA ?PA)
            (not (match-as-part ?NA ?PA))
            (match-as-part ?NA ?)
            (not (match-as-part ?NA ?SS)))
  =>
  (at ?VP (retract ?fact1 ?fact2)
      (assert (view-act world-control-factor ?NA))))

(defrule devide-parts-act-1 "If PA=map(VA)=map(MA),
            [Case1] WCF has Pinit,
                    whole-act(VA)>parts(PA)=x, whole-act(NA)>parts(PA)=y,
                      then PA <- Pinit, map(VA) <- x, map(NA) <- y"
  (declare (salience -220))
  (viewpoint ?VP
            (schema world-control-factor
                    ?fact1 <- (performing-act ?PA)
                    (view-act ?VA)
                    (next-act ?NA)
                    ?fact2 <- (P-init ?Pinit))
            ?fact3 <- (match-as-part ?VA ?PA)
            ?fact4 <- (match-as-part ?NA ?PA)
            (parts ?PA ?x)
            (or (whole-act ?VA ?x)
                (match ?VA ?x)
                (and (match ?VA ?m-va)
                     (or (ako ?m-va ?x)
                         (ako ?x ?m-va))))
            (parts ?PA ?y)
            (or (whole-act ?NA ?y)
                (match ?NA ?y)
                (and (match ?NA ?m-na)
                     (or (ako ?m-na ?y)
                         (ako ?y ?m-NA)))))
  =>
  (at ?VP (sprout (retract ?fact1 ?fact2 ?fact3 ?fact4)
                  (assert (performing-act world-control-factor ?Pinit)
                          (match-as-part ?VA ?x)
                          (match-as-part ?NA ?y)
                          (link ?Pinit ?PA)
                          (undevidable-act world-control-factor ?PA)))))
```

```
(defrule devide-parts-act-2 "If PA=map(VA)=map(MA),
           [Case2] not(WCF has Pinit), PA has parts,
                       then Pinit(WCF) <- search"
  (declare (salience -220))
  (viewpoint ?VP
            (schema world-control-factor
                    (performing-act ?PA)
                    (view-act ?VA)
                    (next-act ?NA)
                    (not (P-init ?)))
            (match-as-part ?VA ?PA)
            (match-as-part ?NA ?PA)
            (parts ?PA ?parts)
            (not (and (condition ?parts (?name ?slot ?value))
                      (not (schema ?name
                                   (?slot ?value)))
                      (not (and (ako ?ako-name ?name)
                                (schema ?ako-name
                                        (?slot ?value))))
                      (not (and (ako ?ako-value ?value)
                                (schema ?name
                                        (?slot ?ako-value))))
                      (not (and (ako ?ako-name ?name)
                                (ako ?ako-value ?value)
                                (schema ?ako-name
                                        (?slot ?ako-value))))))
            (exists (condition ?parts ?)))
  =>
  (at ?VP (sprout (assert (P-init world-control-factor ?parts))
                  (dolist (rest (list*$ (get-schema-value ?PA 'parts)))
                    (unless (equal ?parts rest)
                      (assert (REST-ACT ?PA =rest)))))))

(defrule devide-parts-act-3 "If PA=map(VA)=map(MA),
           [Case3] not(WCF has Pinit) not(PA has parts),
                     or(x=ako(PA) has parts) (x=kind(PA) has parts),
                      then PA <- x, map(VA) <- x, map(NA) <- x"
  (declare (salience -230))
  (viewpoint ?VP
            (schema world-control-factor
                    ?fact1 <- (performing-act ?PA)
                    (view-act ?VA)
                    (next-act ?NA)
                    (not (P-init ?)))
            ?fact2 <- (match-as-part ?VA ?PA)
            ?fact3 <- (match-as-part ?NA ?PA)
            (not (parts ?PA ?))
            (or (ako ?PA ?x)
                (ako ?x ?PA))
            (parts ?x ?))
  =>
  (at ?VP (sprout (retract ?fact1)
                  (retract ?fact2)
                  (retract ?fact3)
                  (assert (performing-act world-control-factor ?x)
                          (match-as-part ?VA ?x)
                          (match-as-part ?NA ?x)))))
```

```
(defrule poison-world-for-order "If PA=map(NA), PA<>map(VA), then poison world"
  (viewpoint ?VP (schema world-control-factor
                        (performing-act ?PA)
                        (next-act ?NA)
                        (view-act ?VA))
            (match-as-part ?NA ?PA)
            (match-as-part ?VA ?)
            (not (match-as-part ?VA ?PA)))
  =>
  (at ?VP (poison "restriction of order")))

(defrule poison-world-for-match-count "If end, match-factor is less, poison world"
  (Viewpoint ?VP
            (schema world-control-factor
                    (matching-phase Phase-2)
                    (similar-service ?SS)
                    (performing-act ?SS))
            (status ?SS done)
            (max-match-factor control-factor ?max-factor))
  (Viewpoint ?VP1
            (schema world-control-factor
                    (match-factor ?factor&:(?factor < ?max-factor))
                    (similar-service ?)))
  =>
  (at ?Vp1
      (poison "This world has a less match-factor than the another world")))

(defrule Make-result-world ""
  (declare (salience 100))
  (Viewpoint ?VP
            (schema world-control-factor
                    (similar-service ?SS)
                    ?fact <- (matching-phase Phase-0)))
  (Viewpoint ?VP1
            (schema world-control-factor
                    (matching-phase Phase-6)
                    (Input-service ?Input)
                    (similar-service ?SS))
            (status ?SS done))
  =>
  (at ?VP (sprout (retract ?fact)
                  (assert (matching-phase world-control-factor Phase-3))
                  (dolist (name (list*$ (get-schema-value ?Input 'parts ?VP)))
                    (retract-schema-value name 'match-as-part)
                    (When (get-schema-value name 'match-as-part ?VP1)
                      (put-schema-value name
                                        'match-as-part
                                        (get-schema-value name 'match-as-part ?VP1)))
                    (When (get-schema-value name 'match-intention ?VP1)
                      (put-schema-value name
                                        'match-intention
                                        (get-schema-value name 'match-intention ?VP1))))))
  (at ?root (assert (match-Phase control-factor find))))
```

```
(defrule find-similar-service ""
  (declare (salience -5000))
  (Viewpoint ?VP1
             (schema world-control-factor
                     ?fact1 <- (matching-phase Phase-2)
                     (Input-service ?Input)
                     (similar-service ?SS)
                     ?fact2 <- (view-act ?VA))
             (status ?SS done)
             (schema control-factor
                     (max-match-factor ?MMF)
                     (act-number ?AN)))
  =>
  (at ?root (assert (schema control-factor
                            (match-Phase find)
                            (max-match-intention-factor =(- ?AN ?MMF)))))
  (at ?VP1
      (retract ?fact1 ?fact2)
      (assert (schema world-control-factor
                      (matching-phase Phase-4)
                      (match-intention-factor =(- ?AN ?MMF))))))

(defrule poison-unrelated-world-prep ""
  (declare (salience -300000))
  =>
  (assert (search-similar-service-Phase-end)))

(defrule poison-unrelated-world ""
  (declare (salience -300000))
  (Viewpoint ?VP
             (search-similar-service-Phase-end)
             (matching-phase world-control-factor Phase-1))
  =>
  (at ?VP (poison "poison unrelated world")))

(undefrule poison-unrelated-world)
```

CLM05:>YOKOTA>art-world>test4-KB.art.2   4/17/90 14:48:57   Page 1

```
;;; -*- Mode: ART; Base: 10.; Package: ART-USER; Syntax: COMMON-LISP -*-

(defschema action ""
  (agent)
  (object)
  (condition)
  (change-from)
  (change-to))

(defschema send-message ""
  (ako action)
  (inverse-act receive-message)
  (agent intelligent-object)
  (message-from intelligent-object)
  (message-to intelligent-object)
  (intention)
  (media)
  (form))

(defschema receive-message ""
  (ako action)
  (inverse-act send-message)
  (agent intelligent-object)
  (message-from intelligent-object)
  (message-to intelligent-object)
  (intention)
  (media)
  (form))

(defschema functional-act ""
  (ako action)
  (input-act receive-message)
  (output-act send-message)
  (agent intelligent-object))

(defschema composite-act ""
  (ako action)
  (parts))

(defschema Place-Call ""
  (aKO Composite-Act)
  (parts Originate-Call Transmit-Tel-NO Alert Answer-Call Talk-Telephone)
  (agent Caller-Human)
  (condition (Caller-Human state Human-Idle)
             (Caller-Telephone state Telephone-onhook)
             (CO-Exchange state CO-Exchange-Idle)
             (Receiver-Human state Human-Idle)
             (Receiver-Telephone state Telephone-onhook)
             (Caller-Human has-A Caller-Telephone)
             (Receiver-Human has-A Receiver-Telephone))
  (change-From (Caller-Human state Human-Idle)
               (Caller-Telephone state Telephone-onhook)
               (CO-Exchange state CO-Exchange-Idle)
               (Receiver-Human state Human-Idle)
               (Receiver-Telephone state Telephone-onhook))
  (change-To (Caller-Human state Human-Talking)
             (Caller-Telephone state Telephone-Offhook)
             (CO-Exchange state CO-Exchange-talking)
             (Receiver-Human state Human-talking)
             (Receiver-Telephone state Telephone-Offhook))
  (Message-From Caller-Human)
  (Message-To Receiver-Human))
```

```
(defschema Originate-Call ""
  (aKO Composite-Act)
  (parts lift-handset-idle Perceive-lift-handset handset-up-to-loop-close close-loop
         Perceive-close-loop-idle close-loop-to-send-dt send-dt-signal receive-voice-signal
         voice-signal-to-voice send-voice listen-DT)
  (agent Caller-Human)
  (condition (Caller-Human state Human-Idle)
             (Caller-Telephone state Telephone-onhook)
             (CO-Exchange state CO-Exchange-Idle)
             (Caller-Human has-A Caller-Telephone))
  (change-From (Caller-Human state Human-Idle)
               (Caller-Telephone state Telephone-onhook)
               (CO-Exchange state CO-Exchange-Idle))
  (change-To (Caller-Human state Human-Listening-DT)
             (Caller-Telephone state Telephone-Offhook)
             (CO-Exchange state CO-Exchange-Sending-DT-Signal)))

(defschema lift-handset ""
  (ako send-message lift)
  (inverse-act Perceive-lift-handset)
  (agent human)
  (object handset)
  (condition (human state human-handset-down)
             (human has-a telephone))
  (change-from (Human state human-handset-down))
  (change-to (Human state human-handset-up))
  (intention Trigger)
  (message-from Human)
  (message-to Telephone)
  (media Object-Moving)
  (form On))

(defschema lift-handset-idle ""
  (aKO lift-handset)
  (agent Caller-Human)
  (condition (Caller-Human state Human-Idle)
             (Caller-Human has-a caller-telephone))
  (change-from (Caller-Human state Human-Idle))
  (message-From Caller-Human)
  (message-To Caller-Telephone))

(defschema Perceive-lift-handset ""
  (aKO Receive-Message)
  (inverse-Act lift-handset)
  (condition (Telephone state Telephone-handset-down)
             (Human has-A Telephone))
  (change-From (Telephone state Telephone-Handset-down))
  (change-To (Telephone state Telephone-Handset-up))
  (intention Trigger)
  (message-From Human)
  (message-To Telephone)
  (media Object-Moving)
  (form On))

(defschema handset-up-to-loop-close ""
  (aKO Functional-Act)
  (input-Act Perceive-Lift-Handset)
  (output-Act Close-Loop)
  (condition (Telephone state Telephone-Handset-up)))

(defschema close-loop ""
  (ako send-message)
  (inverse-act Perceive-close-loop)
  (agent Telephone)
  (object loop)
  (condition (Telephone state Telephone-handset-Up))
  (change-to (Telephone state Telephone-Offhook))
  (intention Trigger)
  (message-from Telephone)
  (message-to CO-Exchange)
  (media Object-Moving)
  (form On))
```

```
(defschema Perceive-close-loop ""
  (ako receive-message)
  (inverse-act close-loop)
  (agent CO-Exchange)
  (object loop)
  (condition (CO-Exchange state loop-open))
  (change-From (CO-Exchange state loop-open))
  (change-To (CO-Exchange state loop-close))
  (intention Trigger)
  (message-from Telephone)
  (message-to CO-Exchange)
  (media Object-Moving)
  (form On))

(defschema Perceive-close-loop-idle ""
  (ako Perceive-close-loop)
  (agent CO-Exchange)
  (object loop)
  (condition (CO-Exchange state caller-loop-open))
  (change-From (CO-Exchange state caller-loop-open))
  (change-To (CO-Exchange state caller-loop-close)))

(defschema close-loop-to-send-dt ""
  (ako functional-act)
  (input-Act Perceive-close-loop-idle)
  (output-Act send-dt-signal)
  (agent CO-Exchange)
  (condition (CO-Exchange state caller-loop-close)))

(defschema send-voice-signal ""
  (ako send-message)
  (inverse-act receive-voice-signal)
  (agent CO-Exchange)
  (object information)
  (intention notice)
  (message-from CO-Exchange)
  (message-to Telephone)
  (media voice-signal))

(defschema send-dt-signal ""
  (ako send-voice-signal)
  (object DT-signal)
  (condition (CO-Exchange state caller-loop-close))
  (change-To (CO-Exchange state CO-Exchange-sending-DT-Signal))
  (form DT))

(defschema receive-voice-signal ""
  (ako receive-message)
  (inverse-act send-voice-signal)
  (agent telephone)
  (object information)
  (condition (telephone state telephone-offhook)))

(defschema voice-signal-to-voice ""
  (ako functional-act)
  (input-Act receive-voice-signal)
  (output-Act send-voice)
  (agent telephone)
  (condition (telephone state telephone-offhook)))

(defschema send-voice ""
  (ako send-message)
  (inverse-act listen)
  (agent telephone)
  (intention notice)
  (message-from Telephone)
  (message-to Human)
  (media voice))

(defschema listen ""
  (ako receive-message)
  (intention notice)
  (media voice))
```

```
(defschema listen-DT ""
  (ako listen)
  (agent Human)
  (object DT)
  (condition (Caller-Human state Human-Handset-Up))
  (change-to (Caller-Human state Human-listening-DT))
  (message-from Telephone)
  (message-to Human)
  (form DT))

(defschema Transmit-Tel-NO ""
  (aKO Composite-Act)
  (condition (caller-human state Human-listening-DT))
  (change-from (caller-human state Human-listening-DT))
  (change-to (caller-human state dial-end)))

(defschema Alert ""
  (aKO Composite-Act)
  (condition (caller-human state dial-end)
             (receiver-human state human-idle))
  (change-from (caller-human state dial-end)
               (receiver-human state human-idle))
  (change-to (caller-human state listening-RBT)
             (receiver-human state listening-Ringing)))

(defschema Answer-Call ""
  (aKO Composite-Act)
  (condition (receiver-human state listening-Ringing))
  (change-from (caller-human state listening-RBT)
               (receiver-human state listening-Ringing))
  (change-to (caller-human state talking)
             (receiver-human state talking)))

(defschema Talk-Telephone ""
  (aKO Composite-Act)
  (condition (caller-human state talking)
             (receiver-human state talking)))



(defschema idv-object ""
  (state))

(defschema intelligent-object ""
  (ako idv-object)
  (has-a))

(defschema human ""
  (ako intelligent-object))

(defschema Caller-Human ""
  (ako human))

(defschema Receiver-Human ""
  (ako human))

(defschema telephone ""
  (ako intelligent-object))

(defschema Caller-Telephone ""
  (ako telephone))

(defschema Receiver-Telephone ""
  (ako telephone))

(defschema CO-Exchange ""
  (ako intelligent-object))
```

```
(defschema States "")

(defschema Human-state ""
  (ako States))

(defschema human-handset-down ""
  (ako Human-state))

(defschema Human-Idle ""
  (ako human-handset-down))

(defschema human-handset-up ""
  (ako Human-state))

(defschema Human-Talking ""
  (ako human-handset-up))

(DEFSCHEMA Human-Listening-DT ""
  (ako human-handset-up))

(defschema Telephone-State ""
  (ako States))

(defschema Telephone-handset-down ""
  (ako Telephone-State))

(defschema Telephone-Handset-up ""
  (ako Telephone-State))

(defschema Telephone-onhook ""
  (ako Telephone-handset-down))

(defschema Telephone-Offhook ""
  (ako Telephone-Handset-up))

(defschema CO-Exchange-State ""
  (ako States))

(defschema loop-Open ""
  (ako CO-Exchange-State))

(defschema loop-close ""
  (ako CO-Exchange-State))

(DEFSCHEMA caller-loop-open ""
  (ako loop-open))

(DEFSCHEMA caller-loop-close ""
  (ako loop-close))

(DEFSCHEMA CO-Exchange-Idle ""
  (ako caller-loop-open))

(DEFSCHEMA CO-Exchange-talking ""
  (ako caller-loop-close))

(DEFSCHEMA CO-Exchange-Sending-DT-Signal ""
  (ako caller-loop-close))
```

```
;;; -*- Mode: Art; Base: 10.; Package: Art-user; Syntax: Common-lisp -*-

(assert (schema act1
                (match lift-handset)
                (post-act act2))
        (schema act2
                (match close-loop)
                (pre-act act1)
                (post-act act3))
        (schema act3
                (match send-voice-signal)
                (pre-act act2)
                (post-act act4))
        (schema act4
                (match talk-telephone)
                (pre-act act3))
        (schema control-factor
                (act-number 4))
        (schema world-control-factor
                (Input-service act5))
        (schema act5
                (parts act1 act2 act3 act4)))

(assert (schema act1
                (match lift-handset)
                (post-act act2))
        (schema act2
                (match send-voice-signal)
                (pre-act act1)
                (post-act act3))
        (schema act3
                (match close-loop)
                (pre-act act2)
                (post-act act4))
        (schema act4
                (match talk-telephone)
                (pre-act act3))
        (schema control-factor
                (act-number 4))
        (schema world-control-factor
                (Input-service act5))
        (schema act5
                (parts act1 act2 act3 act4)))
```

```
;;; -*- Mode: ART; Base: 10.; Package: ART-USER; Syntax: COMMON-LISP -*-

(defrule set-init-act ""
  (declare (salience 50))
  (Viewpoint ?VP
           (schema world-control-factor
                   (matching-phase Phase-1)
                   (not (init-act ?))
                   (performing-act ?PA)
                   (view-act ?VA))
           (or (not (match-as-part ?VA ?PA))
               (and (match-as-part ?VA ?PA)
                    (next-act world-control-factor ?NA)
                    (match-as-part ?NA ?)
                    (not (match-as-part ?NA ?PA)))))
  =>
  (at ?VP (assert (init-act world-control-factor ?PA))))

(defrule set-last-act ""
  (declare (salience 50))
  (Viewpoint ?VP
           (schema world-control-factor
                   (init-act ?)
                   (not (last-act ?))
                   (performing-act ?PA))
           (not (rest-act ? ?)))
  =>
  (at ?VP (assert (last-act world-control-factor ?PA))))

(defrule set-initial-control-act-for-intention-match ""
  (declare (salience 100))
  (Viewpoint ?VP
           (schema world-control-factor
                   (matching-phase Phase-4)
                   (input-service ?IS)
                   (not (view-act ?))
                   (init-act ?INIT))
           (part-of ?Act ?IS)
           (not (pre-act ?Act ?)))
  =>
  (at ?VP (assert (schema world-control-factor
                          (view-act ?Act)
                          (pre-act ?INIT)))))

(defrule next-view-act-for-intention-match ""
  (declare (salience 100))
  (Viewpoint ?VP
           (schema world-control-factor
                   (match-intention-factor ?MIF)
                   ?fact1 <- (view-act ?VA)
                   ?fact2 <- (pre-act ?)
                   ?fact3 <- (matching-phase Phase-4))
           (max-match-intention-factor control-factor ?MIF)
           (match-as-part ?VA ?MAP))
  (case
    ((Viewpoint ?VP (next-act world-control-factor ?))
     =>
     (at ?VP (poison "not match")))
    ((Viewpoint ?VP (post-act ?VA ?POST))
     =>
     (at ?VP (retract ?fact1 ?fact2)
         (assert (view-act world-control-factor ?POST)
                 (pre-act world-control-factor ?MAP))))
    (otherwise
      => (retract ?fact3)
      (at ?VP (assert (matching-phase world-control-factor Phase-6))))))
```

```
(defrule set-next-act-for-intention-match ""
  (declare (salience 100))
  (Viewpoint ?VP
             (schema world-control-factor
                     ?fact1 <- (match-intention-factor ?MIF)
                     ?fact2 <- (view-act ?VA)
                     (not (next-act ?)))
             (max-match-intention-factor control-factor ?MIF)
             (not (match-as-part ?VA ?)))
  (case
    ((Viewpoint ?VP (post-act ?VA ?post))
     =>
     (at ?VP (sprout (assert (next-act world-control-factor ?post)))
         (sprout (retract ?fact1 ?fact2)
                 (assert (schema world-control-factor
                                 (view-act ?post)
                                 (match-intention-factor =(- ?MIF 1)))))))
    ((Viewpoint ?VP (schema world-control-factor
                            (last-act ?last)
                            ?fact3 <- (matching-phase Phase-4)))
     =>
     (at ?VP (sprout (assert (next-act world-control-factor ?last)))
         (sprout (retract ?fact1 ?fact3)
                 (assert (schema world-control-factor
                                 (match-intention-factor =(- ?MIF 1))
                                 (matching-phase Phase-6))))))))

(defrule reset-next-act-for-intention-match ""
  (declare (salience 100))
  (Viewpoint ?VP
             (schema world-control-factor
                     (matching-phase Phase-4)
                     (view-act ?VA)
                     ?fact1 <- (next-act ?NA)
                     (input-service ?IS))
             (parts ?IS ?NA)
             (or (match-as-part ?NA ?MAP)
                 (and (not (match-as-part ?NA ?))
                      (or (post-act ?NA ?MAP)
                          (and (not (post-act ?NA ?))
                               (last-act world-control-factor ?MAP))))))
  =>
  (at ?VP (sprout (retract ?fact1)
                  (assert (next-act world-control-factor ?MAP)))))
```

```
(defrule hyp-get-match-intention-phase-1 ""
  (declare (salience -1000))
  (viewpoint ?VP-NOW
             (schema world-control-factor
                     ?fact1 <- (matching-phase Phase-4)
                     ?fact2 <- (pre-act ?PRE)
                     (input-service ?IS)
                     (view-act ?VA)
                     ?fact3 <- (next-act ?POST))
             (not (parts ?IS ?POST))
             (not (match-as-part ?VA ?)))
  (viewpoint ?VP-POST
             (performing-act world-control-factor ?POST))
  (test (vp-inherits-from? ?VP-NOW ?VP-POST))
  (viewpoint ?VP-THEN
             (performing-act world-control-factor ?ACT)
             (match-intention ?ACT ?int)
             (or (match-intention ?VA ?int)
                 (not (match-intention ?VA ?))))
  (test (and (vp-inherits-from? ?VP-POST ?VP-THEN)
             (not (vp-equal? ?VP-POST ?VP-THEN))))
  (viewpoint ?VP-PRE
             (performing-act world-control-factor ?PRE))
  (test (and (vp-inherits-from? ?VP-THEN ?VP-PRE)
             (not (vp-equal? ?VP-THEN ?VP-PRE))))
  =>
  (at ?VP-NOW
      (sprout (retract ?fact1 ?fact2 ?fact3)
              (assert (schema world-control-factor
                               (pre-act ?ACT)
                               (matching-phase sub-phase-1)
                               (match-intention-act ?ACT))
                      (match-intention ?VA ?ACT)
                      (match-as-part ?VA ?ACT)))))

(defrule hyp-get-match-intention-phase-2 ""
  (declare (salience 1000))
  (viewpoint ?VP
             (schema world-control-factor
                     ?fact <- (matching-phase sub-phase-1)
                     (match-intention-act ?ACT)
                     (view-act ?VA))
             (message-from ?ACT ?from)
             (or (message-from ?VA ?from)
                 (not (message-from ?VA ?))
                 (and (message-from ?VA ?from1)
                      (or (ako ?from ?from1)
                          (ako ?from1 ?from)))))
  =>
  (at ?VP (retract ?fact)
      (assert (matching-phase world-control-factor sub-phase-2))))

(defrule hyp-get-match-intention-phase-3 ""
  (declare (salience 1000))
  (viewpoint ?VP
             (schema world-control-factor
                     ?fact <- (matching-phase sub-phase-2)
                     (match-intention-act ?ACT)
                     (view-act ?VA))
             (message-to ?ACT ?to)
             (or (message-to ?VA ?to)
                 (not (message-to ?VA ?))
                 (and (message-to ?VA ?to1)
                      (or (ako ?to ?to1)
                          (ako ?to1 ?to)))))
  =>
  (at ?VP (retract ?fact)
      (assert (matching-phase world-control-factor sub-phase-3))))
```

```
(defrule hyp-get-match-intention-phase-4 ""
  (declare (salience 1000))
  (viewpoint ?VP
             (schema world-control-factor
                     ?fact1 <- (matching-phase sub-phase-3)
                     (match-intention-act ?ACT)
                     ?fact2 <- (view-act ?VA))
             (agent ?ACT ?age)
             (or (agent ?VA ?age)
                 (not (agent ?VA ?))
                 (and (agent ?VA ?age1)
                      (or (ako ?age ?age1)
                          (ako ?age1 ?age)))))
  =>
  (at ?VP (retract ?fact1 ?fact2)
      (assert (matching-phase world-control-factor phase-4))))

(defrule devide-act-for-match-intention ""
  (declare (salience -1000))
  (viewpoint ?VP-NOW
             (schema world-control-factor
                     ?fact1 <- (matching-phase phase-4)
                     ?fact2 <- (pre-act ?PRE)
                     (input-service ?IS)
                     (view-act ?VA)
                     ?fact3 <- (next-act ?POST))
             (not (parts ?IS ?POST))
             (not (match-as-part ?VA ?)))
  (viewpoint ?VP-POST
             (performing-act world-control-factor ?POST))
  (test (vp-inherits-from? ?VP-NOW ?VP-POST))
  (viewpoint ?VP-THEN
             ?fact4 <- (performing-act world-control-factor ?ACT)
             ?fact5 <- (link ?act ?)
             (or (parts ?ACT ?parts)
                 (and (ako ?ACT ?XX)
                      (parts ?XX ?parts))
                 (and (ako ?XX ?ACT)
                      (parts ?XX ?parts)))
             (not (and (condition ?parts (?name ?slot ?value))
                       (not (schema ?name
                                    (?slot ?value)))
                       (not (and (ako ?ako-name ?name)
                                 (schema ?ako-name
                                         (?slot ?value))))
                       (not (and (ako ?ako-value ?value)
                                 (schema ?name
                                         (?slot ?ako-value))))
                       (not (and (ako ?ako-name ?name)
                                 (ako ?ako-value ?value)
                                 (schema ?ako-name
                                         (?slot ?ako-value))))))
             (exists (condition ?parts ?)))
  (viewpoint ?VP-NOW (not (undevidable-act world-control-factor ?ACT)))
  (test (and (vp-inherits-from? ?VP-POST ?VP-THEN)
             (not (vp-equal? ?VP-POST ?VP-THEN))))
  (viewpoint ?VP-PRE
             (performing-act world-control-factor ?PRE))
  (test (and (vp-inherits-from? ?VP-THEN ?VP-PRE)
             (not (vp-equal? ?VP-THEN ?VP-PRE))))
  =>
  (at ?VP-NOW
      (sprout (retract ?fact1 ?fact2 ?fact3); ?fact6)
              (assert (schema world-control-factor
                              (pre-act ?ACT)
                              (matching-phase Phase-5)))))
  (at ?vp-THEN
      (sprout (retract ?fact4 ?fact5)
              (assert (performing-act world-control-factor ?parts)
                      (link ?parts ?ACT))
              (dolist (rest (list*$ (get-schema-value ?ACT 'parts)))
                (unless (equal ?parts rest)
                  (assert (REST-ACT ?ACT =rest)))))))

;;; (defrule set-next-act-parts-end-for-get-match-intention "")
```