〔非公開〕

ＴＲ－Ｃ－００５３

ネットワーク仕様記述言語処理系仕様書

林　　潔
KIYOSHI　HAYASHI

１９９０．８．７

ＡＴＲ通信システム研究所

ネットワーク仕様処理系

仕様書

1990年 8月

目次

1　概要

本システムでは、「ネットワーク仕様記述法」により作成された仕様データを評価する「ネットワーク仕様記述言語処理系」を実現する。

ネットワーク仕様記述法（並列処理による部品化仕様記述法）とは、通信システムを複数プロセスの協調処理として仕様化する場合に、システム動作の起動トリガとなるイベントに着目し、プロセスの仕様を次の3種類の制約条件に分けて仕様化するものである。

①イベントの分岐条件の定義（分配制約）
②プロセス個別機能の動作定義（機能制約）
③外部プロセスとの同期条件の定義（同期制約）

本システムは、仕様データ（定数定義式、変数定義式および制約定義式）を読み込んで、制約を解釈実行する。

2　システム概念

本システムの概念図を「図2.1 システム概念図」に示す。

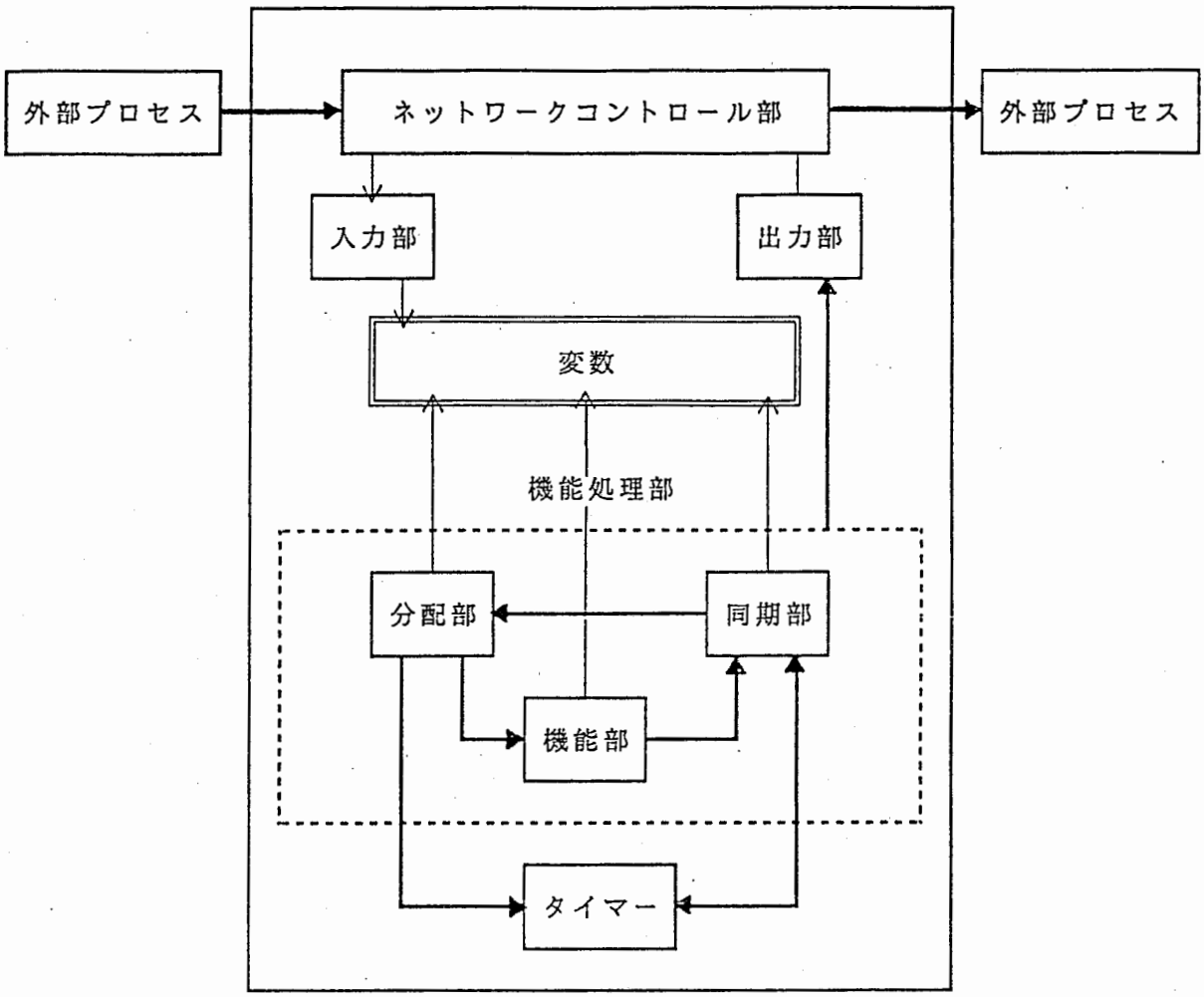


図2.1 システム概念図

3　処理系の機能構成

処理系の機能構成図を「図3.1 処理系機能構成図」に示す。

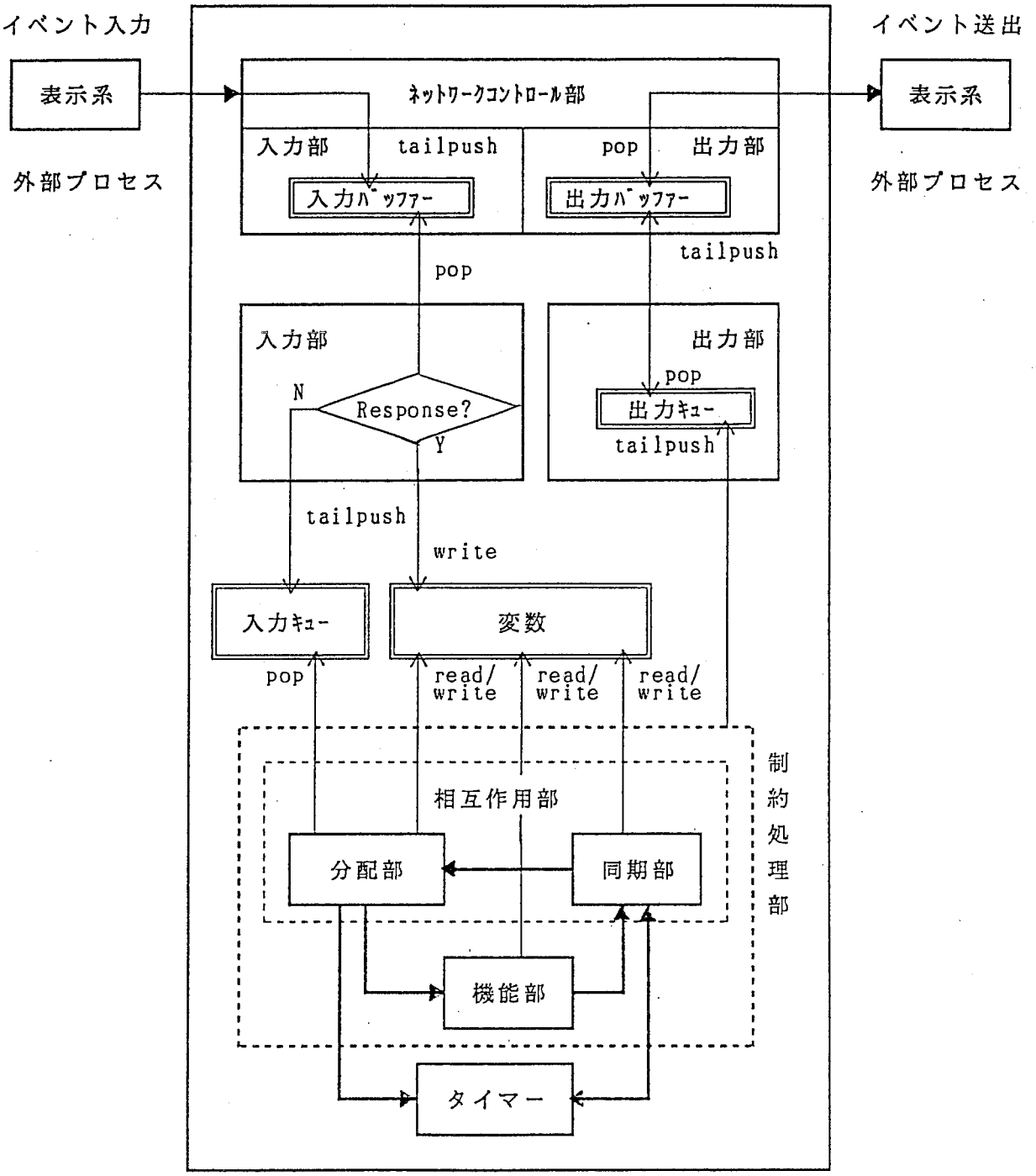


図3.1 処理系機能構成図

4　制約処理部

4－1　基本制約処理

制約式の評価は、条件部が真になる制約式のボディ部を評価することにより行われる。

4－1－1　条件部の判定

条件部には複数の条件式が記述される。

条件式の種類は以下の通りである。

①イベント条件式
②変数条件式
③ＡＮＤ条件式
④ＯＲ条件式

(1)イベント条件式の判定

イベント変数の送出元および値の判定を行う。

条件式の送出元（および値が記述されていればその値）が、そのイベント変数インスタンスにセットされている送出元（および値）と一致する場合、その条件式は成立する。

送出元（あるいは値）に一時変数が記述されていた場合、その一時変数の値にはそのイベント変数の送出元（あるいは値）をセットする。

一旦一時変数に値がセットされた後は、その一時変数はその値として処理する。

（例１）

```
現在の実行部　：分配部
イベント条件式：( イベント変数Ａ　@送出元ａ　値1 )
```

〈 イベント変数Ａ 〉インスタンス（その１）

送出元	送出元ａ	
値	値1	
実行部	(:dcmp :func)	→ 成立

〈 イベント変数Ａ 〉インスタンス（その２）

送出元	送出元ｂ	
値	値1	
実行部	(:dcmp :func)	→ 成立しない

〈 イベント変数Ａ 〉インスタンス（その３）

送出元	送出元ａ	
値	値2	
実行部	(:dcmp :func)	→ 成立しない

（例２）

```
現在の実行部　：分配部
イベント条件式：( イベント変数Ａ　送出元ａ　*var )
```

〈 イベント変数Ａ 〉インスタンス

送出元	送出元ａ	
値	値1	
実行部	(:dcmp :func)	→ 成立 一時変数〈 var 〉の値には、 〈 値1 〉がセットされる。

(2)変数条件式の判定

変数（イベント変数以外）の値の判定を行う。

条件式の値が、その変数インスタンスにセットされている、現在実行している部（分配部、機能部あるいは同期部）に対応するＶＩＥＷと組になっている値と一致する場合にその条件式は成立する。

但し、条件否定が記述されていた場合は、その変数のもつ値が条件式に記述された値と等しければ条件式は成立しないものとする。

値に一時変数が記述されていた場合、その一時変数の値にはそのイベント変数の値をセットする。

一旦一時変数に値がセットされた後は、その一時変数はその値として処理する。

（例１）

```
現在の実行部　：分配部
変数条件式　　：( $状態変数Ｂ　値1 )
```

〈 状態変数Ｂ 〉インスタンス（その１）

ＶＩＥＷ　(:dcmp　:func)

値　　(値1　　機能部の値)　→　成立

〈 状態変数Ｂ 〉インスタンス（その２）

ＶＩＥＷ　(:dcmp)

値　　(値2　　)　→　成立しない

（例２）

```
現在の実行部　：分配部
変数条件式　　：( ~ $状態変数Ｂ　値1 )
```

〈 状態変数Ｂ 〉インスタンス（その１）

ＶＩＥＷ　(:dcmp)

値　　(値1　　)　→　成立しない

〈 状態変数Ｂ 〉インスタンス（その２）

ＶＩＥＷ　(:dcmp)

値　　(値2　　)　→　成立

（例３）

```
現在の実行部　：分配部
変数条件式　　：( $状態変数Ｂ　*var　)
```

〈 状態変数Ｂ 〉インスタンス

ＶＩＥＷ　　(:dcmp)

値　　(値1　　)　　→　成立

一時変数〈 var 〉の値には〈 値1 〉がセットされる。

(3)ＡＮＤ条件式の判定

ＡＮＤ条件式に記述された全条件式（イベント条件式、変数条件式、ＡＮＤ条件式あるいはＯＲ条件式）を１つずつ判定し、全ての条件式が満たされた場合に成立する。

(4)ＯＲ条件式の判定

ＯＲ条件式に記述された全ての条件式（イベント条件式、変数条件式、ＡＮＤ条件式あるいはＯＲ条件式）を１回評価し、そのうち１つでも条件が成立している条件式があった場合に成立する。

4－1－2　ボディ部の実行

ボディ部には複数のボディ式があり、ボディ部の評価はその制約式の全ボディ式を順に評価することにより行われる。

ボディ式の種類は以下の通りである。

①イベント定義式
②状態定義式
③タイマー解除式

(1)イベント定義式の評価

①イベント（送出先の内部・外部を問わない）を 表示系 に送るために出力キューにイベント出力情報入れる。*1

②自プロセスにイベントを送出する場合は、以下の処理を行う。
機能部へ送出：そのまま値と送出元をセットする。
同期部へ送出：値と送出元を同じ順番で値スロット、送出元スロットに入れる。

（例）
〈 イベント変数Ａ 〉に対し、〈 %@BC 〉から〈 値２ 〉が送られていて、かつ同変数に機能部から同期部に〈 値１ 〉を送出する場合

〈 イベント変数Ａ 〉

送出元　→（ %@BC)
値　→（ 値２）
実行部　→（ :supr ）

↓

送出元　→（ :func %@BC)
値　→（ 値１　値２）
実行部　→（ :supr ）

○別のプロセスへイベントを送出するイベント定義式があった場合、ボディ部評価後自プロセスのシリアルナンバーを再設定する。

*1 出力キューにイベント送出式を入れるのはルールの実行部処理の中で行う。
このとき、メッセージid（メッセージ-プロセスid、メッセージ-シリアルNo）の設定、およびタイマー値設定も行う。

メッセージid の設定については「(4)メッセージidの設定」参照のこと。
タイマー値設定については「(5)タイマー値の設定」参照のこと。
イベント出力情報については「６－２　出力情報」参照のこと。

(2)状態定義式の評価

状態変数に値をセットする。

値を、対応するＶＩＥＷと同じ位置にあった値と置き換える。

（例）

分配部（:dcmp）で、＜状態変数Ｂ＞に、＜値３＞をセットする場合

＜状態変数Ｂ＞

ＶＩＥＷ　→（:dcmp :func)

値　　→（値４　値５）

↓

ＶＩＥＷ　→（:dcmp :func)

値　　→（値３　値５）

(3)タイマー解除式の評価

タイマーを解除する。

「７－２　分配部および同期部におけるタイマー処理」を参照のこと。

(4)メッセージidの設定

「メッセージid」とは、受信イベントが自プロセスへの新たなイベントなのかあるいは、自プロセスへの応答なのかを判定するために設定するものである。
メッセージid は、イベントを発行したプロセスのid と、そのプロセスのそのときのシリアルNo からなる。

送出情報のメッセージidは、そのイベント送出が新たなイベントの発行かあるいは返答かの判断をしたうえで、以下の通りに設定する。

①送出要求元が分配部の場合、新たなイベントの発行とする。

メッセージid:
(自分のプロセスid　自プロセスで設定するシリアルNo)

②同期部からきている場合で
i)そのイベント変数の送出先が、分配部でセットした送出元と等しい場合（応答の場合）、その送出元への返答とする。

メッセージid:
受け取ったメッセージidをそのまま返す。

ii)等しくない場合（イベントの分配の場合）、新たなイベントの発行とする。

メッセージid:
(自分のプロセスid　自プロセスで設定するシリアルNo)

従って、例えばプロセスＡからプロセスＢへ新たにイベントを発行した場合のイベントの送出時の情報は次のようになる。

```
(イベント変数名1  値1              ; イベント情報
 プロセスB-id     unbound          ; 送出先情報
 プロセスA-id     supr             ; 送出元情報
 プロセスA-id     シリアルNo1      ; メッセージid
 100                          )   ; タイマー情報
```

また逆に、プロセスＢからプロセスＡに返答する場合には、プロセスＢが受け取ったメッセージidをそのまま付けて返す。
イベント送出時の情報は次のようになる。

```
(イベント変数名2   値2              ; イベント情報
 プロセスA-id      unbound          ; 送出先情報
 プロセスB-id      supr             ; 送出元情報
 プロセスA-id      シリアルNo1      ; メッセージid
 0                              )   ; タイマー情報
```

(5)タイマー値の設定

外部プロセスにイベントを送るときに、タイマー設定式が記述されていれば、イベント送出情報のタイマー値にタイマー時間をセットする。
タイマー設定式が記述されていない場合はタイマー値に０をセットする。

(6)制約式評価後のイベント変数

条件部にイベント条件式が記述されていて、かつその条件式が成立した場合、現在実行している部によって処理が異なる。

①現在の実行部が分配部あるいは機能部：
条件式で評価したイベント変数の値と送出元は不定に戻す。

②現在の実行部が同期部：
条件式が真になった値とそれに対応する送出先のみをそのイベント変数インスタンスのスロットから除く。
この時値が何もなくなれば（ nil になった場合）、変数の値と送出元を不定に戻す。

4－2　分配部

分配制約評価処理を以下に示す。

①入力待の状態[*1]となる。
②入力待の状態から抜けると、分配部の評価を開始する。
　②-0 イベント送出情報の送出元、メッセージ-シリアルNoおよびメッセージ-プロセスid をグローバル変数にセットする。
　②-1 自プロセスのシリアルナンバーを設定する。
　②-2 分配部の先頭から条件式が真になる分配制約をチェックする。
　②-3 分配部の最後までチェックした段階で
　　i)条件式が真になる分配制約があった場合
　　その制約式を評価する。
　　制御を機能部に移す。
　　ii)条件式が真になる分配制約がなかった場合
　　この回調べたイベント変数を不定に戻し①から分配制約評価処理を繰り返す。

[*1]　ここでいう『入力待の状態』というのは、次の状態を意味する。
即ち、入力キューに何も入っていない場合（ex.初期状態）にイベント変数に値のセット要求がくるまで待つ状態である。
　入力キューにイベント変数が入っていた場合は、最初にキューに入ったイベント変数をキューから取りだし値をセットする。この場合入力待の状態に入っても実際待つことなしにすぐ抜けることになる。

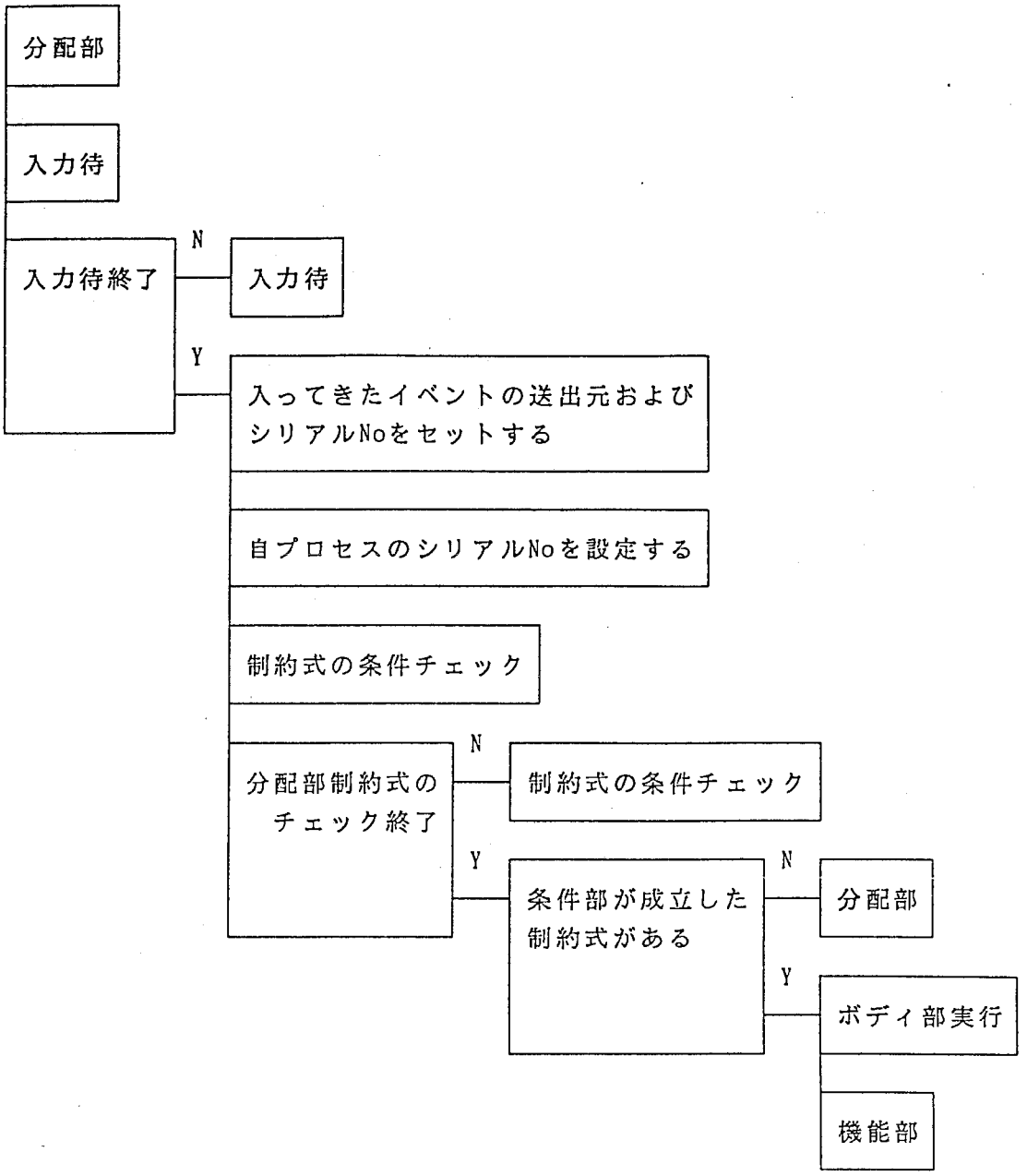

分配部
入力待
入力待終了
N
入力待
Y
入ってきたイベントの送出元および
シリアルNoをセットする
自プロセスのシリアルNoを設定する
制約式の条件チェック
分配部制約式の
チェック終了
N
制約式の条件チェック
Y
条件部が成立した
制約式がある
N
分配部
Y
ボディ部実行
機能部

図4.1 分配部処理

4－3　機能部

機能制約評価処理を以下に示す。

①条件式が真になる機能制約をチェックする。
②機能部の最後までチェックした段階で条件式が真になる機能制約があった場合、制約式を評価する。
③制御を同期部に移す。

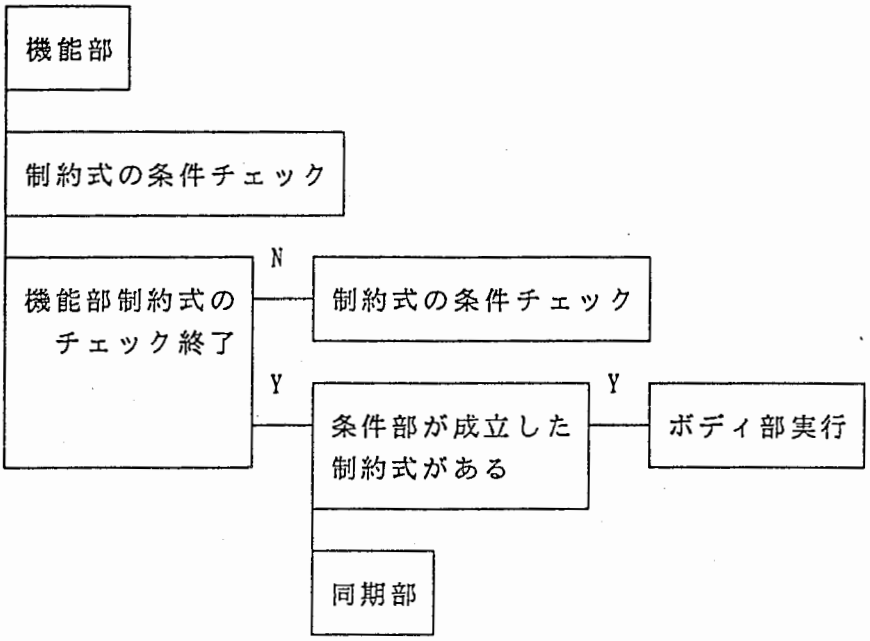


図4.2 機能部処理

4－4　同期部

(1)同期部制約評価処理

同期制約評価処理を以下に示す。

①条件式が真になる同期制約を評価する*1*2。
②同期部で使用されているイベント変数で、値がバインドされている変数がなくなるまで、同期制約の評価を繰り返す。
③同期部で使用されているイベント変数で、値がバインドされているイベント変数がなくなった段階で
　　i)機能部へイベントを送る必要がある場合、制御を機能部へ移す。
　　ii)機能部へ送る必要はない場合、制御を分配部へ移す。

○同期部から別の部へ制御を移す際に、実行されているタイマープロセスがあればタイマーを中断する。

*1　同期部では、自プロセスと外部プロセスが同期を取る場合の制約を記述する。
そのため、イベントを受信した場合、新たな入力イベントなのか、自プロセスが外部プロセスへ送出したイベントへの応答であって時間内に返ってきたイベントなのかどうかの判定をしてから適当な処理をする必要がある。
応答待ちイベントについては「実行リスト」で管理する。

イベント受信時の判定については「5　入力部」を参照のこと。
実行リストの管理については「(2)応答待ち状態（実行リスト）の管理」を参照のこと。

*2　同期部を実行している間、別のプロセスまたは現在実行している部（同期部）のボディ部の評価によりイベント変数、状態変数、あるいはアドレス変数に値がセットされた場合、その変数も別の制約式の評価条件として用いる。

(2)応答待ち状態（実行リスト）の管理

外部プロセスへイベントを送出した場合、その応答の待ち状態にあるかどうかの判定を、「実行リスト」で管理する。

実行リストは、以下のように、その要素が送出時に付与したメッセージidと送出先をもつリストである。

メッセージidを含むイベント送出の際に出力する情報については「６　出力部」を参照のこと。

実行リスト ＝（（ メッセージid　送出先 ）（ メッセージid　送出先 ）・）
＝（（（ 自プロセスid　シリアルNo ）送出先 ）・・）

①実行リストへの追加

外部プロセスへイベントを送出したときに実行リストに追加する。

②実行リストからの削除

次の場合に実行リストから該当するものを削除する。

i)イベントを送ったプロセスから応答がきた場合

ii)送ったイベントがタイムアウトになった場合

③実行リストのリセット

実行リストは、処理が同期部から分配部または機能部へ移った時点でリセットされる。

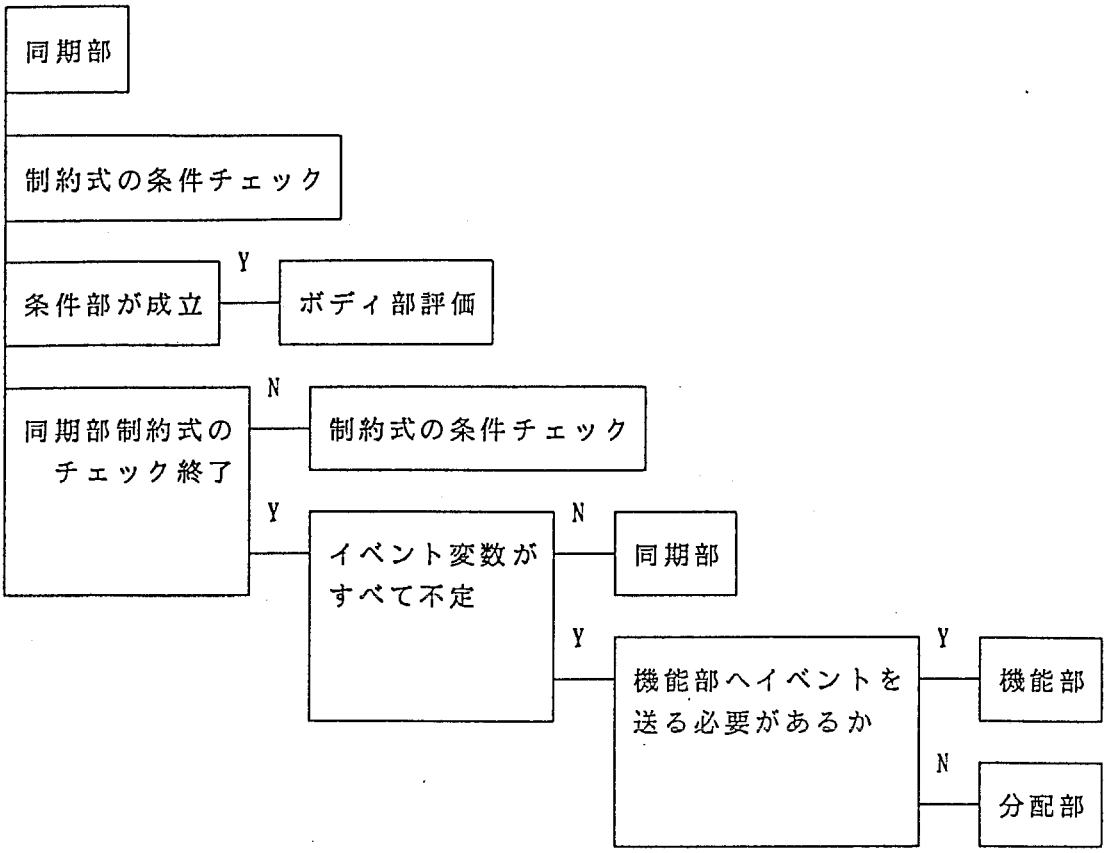

同期部
制約式の条件チェック
条件部が成立
Y
ボディ部評価
同期部制約式の
チェック終了
N
制約式の条件チェック
Y
イベント変数が
すべて不定
N
同期部
Y
機能部へイベントを
送る必要があるか
Y
機能部
N
分配部

図4.3 同期部処理

5　入力部

5－1　イベント受信時のキューイング／セット処理

外部プロセスからイベントを受け取った場合、ネットワーク入力部が受け取ったイベント情報を入力バッファーに入れる。

入力部では、入力バッファーを調べ、イベント情報が入っていた場合１つとりだしそのイベントを「入力イベント」としてキューイングするか、「応答待ちイベント」としてイベント変数のセットを行うか判定して適当な処理を行う。

判定は以下のように行う。

①受信イベントのプロセスidが自プロセスidと等しくなければ④へ。
　等しければ②へ。

②受信イベントのメッセージ-シリアルNoが実行リスト中にあれば③へ。
　なければその情報は捨てる。

③受信イベントの送出元と、実行リストの該当した要素の「送出先」が等しければ、応答待ちイベントとしてイベント変数のセットを行う。
　等しくなければその情報は捨てる。

④入力イベントとしてキューイングする。

③でイベント変数へのセットが行われた場合には、実行リスト中からその要素を削除する。

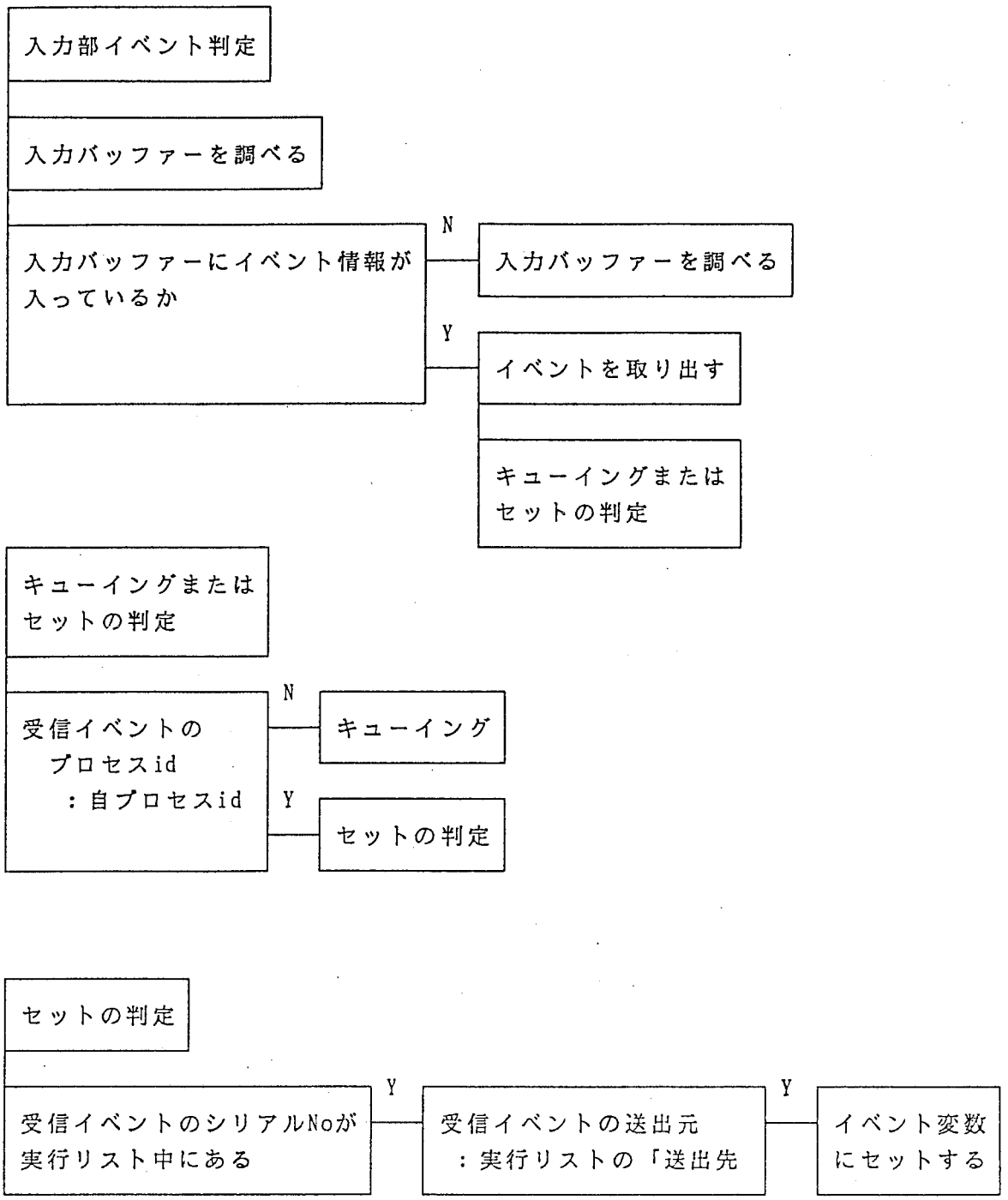


図5.1 入力部イベント判定

5－2　入力情報

入力情報は以下の通りである。

(EVENT　イベント変数〈 送出先情報 〉〈 送出元情報 〉〈 メッセージid 〉
〈 タイマー値〉〈 値 〉*)

〈 送出先情報 〉
::=〈 送出先のプロセスid 〉〈 プロセスブロック 〉

〈 送出元情報 〉
::=〈 送出元のプロセスid 〉〈 プロセスブロック 〉

〈 メッセージid 〉
::=〈 メッセージ-プロセスid 〉〈 メッセージ-シリアルNo 〉

〈 タイマー値 〉
::= タイムアウトまでの時間または0

〈 送出先のプロセスid 〉
::= イベント送出先のプロセスの識別子

〈 送出元のプロセスid 〉
::= イベント送出元のプロセスの識別子

〈 メッセージ-プロセスid 〉
::= 受け渡されるイベントに付与された識別子

〈 メッセージ-シリアルNo 〉
::= 受け渡されるイベントに付与されたシリアルナンバー

〈 値 〉
::= 受け渡されるイベントの値

〈 プロセスブロック 〉
::= dcmp | func | supr | unknown　　*1

* 0個以上任意個

*1 自プロセスの分配部の場合　dcmp
〃　　機能部の場合　func
〃　　分配部の場合　supr
別のプロセスの場合　unknown

6　出力部

6－1　出力処理

制約処理部は出力キューに送出イベント情報を入れることで出力要求を出す。

出力部では出力キューを調べ、イベント情報が入っていた場合１つとりだしネットワーク出力部の出力バッファーに入れる。このとき、イベント情報の先頭に表示系へのストリームを追加する。

また出力情報を分析し、以下の該当する処理を行う。

①外部プロセスへのイベント送出の場合、実行リストに情報を追加する。

②タイマーが設定されていた場合、タイマーを起動する。

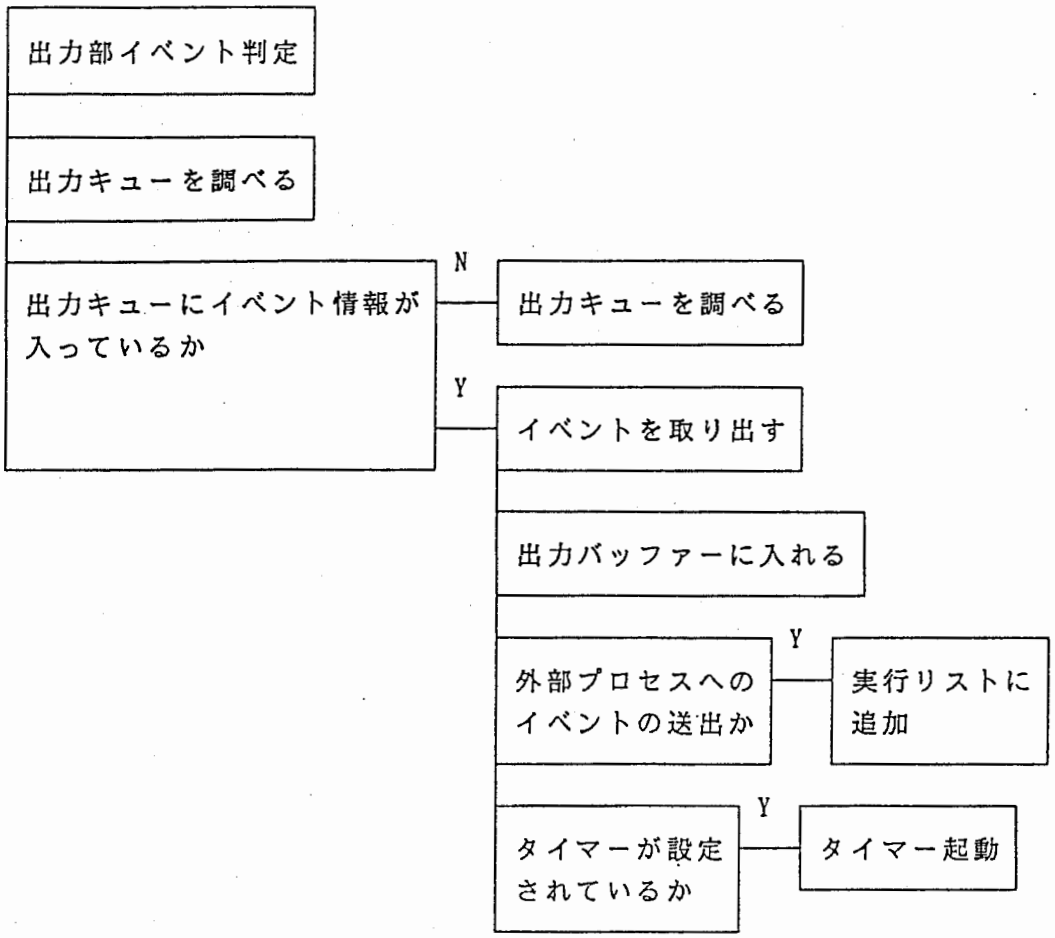


図6.1 出力部イベント判定

6－2　出力情報

出力情報は以下の通りである。

(EVENT　イベント変数〈 送出先情報 〉〈 送出元情報 〉〈 メッセージid 〉
〈 タイマー値〉〈 値 〉*)

〈 送出先情報 〉
::=〈 送出先のプロセスid 〉〈 プロセスブロック 〉

〈 送出元情報 〉
::=〈 送出元のプロセスid 〉〈 プロセスブロック 〉

〈 メッセージid 〉
::=〈 メッセージ-プロセスid 〉〈 メッセージ-シリアルNo 〉

〈 タイマー値 〉
::= タイムアウトまでの時間または0

〈 送出先のプロセスid 〉
::= イベント送出先のプロセスの識別子

〈 送出元のプロセスid 〉
::= イベント送出元のプロセスの識別子

〈 メッセージ-プロセスid 〉
::= 受け渡されるイベントに付与された識別子

〈 メッセージ-シリアルNo 〉
::= 受け渡されるイベントに付与されたシリアルナンバー

〈 値 〉
::= 受け渡されるイベントの値

〈 プロセスブロック 〉
::= dcmp | func | supr | unknown　　*1

* 0個以上任意個

*1 自プロセスの分配部の場合　dcmp
〃　　機能部の場合　func
〃　　分配部の場合　supr
別のプロセスの場合　　unknown

7　タイマー処理部

7－1　タイマー基本処理

タイマーが設定された場合、時間を計る「タイマー関数」を別プロセスで起動する。
タイマーは、「タイマーリスト」で管理する。
タイマーリストは以下のようにイベント送出情報およびタイマーで付加したタイマープロセスidをもつリストである。

イベント送出情報については「６－２　出力情報」を参照のこと。

タイマーリスト
　　::=（〈 タイマー情報 〉* ）

〈 タイマー情報 〉
　　::=（ イベント送出情報 〈 タイマープロセスid 〉）

〈 タイマープロセスid〉
　　::= 起動したタイマー関数のプロセスid

* ０個以上任意個

(1)タイマー起動

イベント送出情報のタイマー値が０でない場合、時間を計る別プロセスを起動する。
起動したプロセスidをタイマープロセスidとしてセットする。
タイマー起動時に受け取ったイベント送出情報と、タイマープロセスidをリストにしてタイマーリストに入れる。

(2)タイマー正常終了（タイムアウトになった場合）

設定時間がきた場合、タイムアウトイベントを発生させる。
すなわち自プロセスである :@timer から同期部に対して、送出先（タイミングを取る相手）とメッセージidのリストをパラメータとするTimeoutイベントをセットする。

タイマーリストから、該当する要素を削除する。

(3)タイマー中断（ReleaseTimig イベントが発行された場合）

タイマーリストから、送出先をキーとして該当する要素を探し、そのプロセスを切る。（タイマープロセスidに :kill を送る。）

タイマーリストから、該当する要素を削除する。

７－２　分配部および同期部におけるタイマー処理

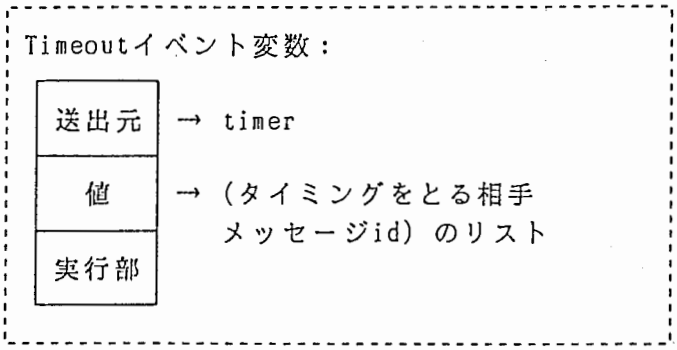
(1)タイマー設定

制約処理部のルール実行部処理において、外部プロセスにイベントを送るときに、タイマー設定式が記述されていれば、イベント送出情報のタイマー値にタイマー時間をセットする。

タイマー設定式が記述されていない場合はタイマー値を０にする。

(2)タイムアウトイベント発生時

i)イベント変数 Timeout の値スロットに「タイミングをとる相手」と「メッセージid」のリストを追加する。
ii)「実行リスト」から、該当する要素を削除する。



(例) 発生したイベントの
メッセージid : (p-1 id-#5)
タイミングをとる相手 : @BC2

実行idリスト : (((p-1 id-#5) @BC2)((p-1 id-#4) @cwm))
Timeoutイベント変数 :

送出元	→ (timer)
値	→ ((@BC1 p-1 id-#4))
実行部	

↓

実行リスト : (((p-1 id-#4) @cwm))
Timeoutイベント変数 :

送出元	→ (timer timer)
値	→ ((@BC2 p-1 id-#5) (@BC1 p-1 id-#4))
実行部	

(3)タイマー設定解除

タイマー設定解除式で、解除したいタイマーのタイミングを取る相手を指定する。
「実行リスト」から、該当する要素を削除する。

（例）タイミングをとる相手　：　@DEST

「実行idリスト」　：　(((p-1 id-#5) @DEST) ((p-1 id-#4) @BC))

↓

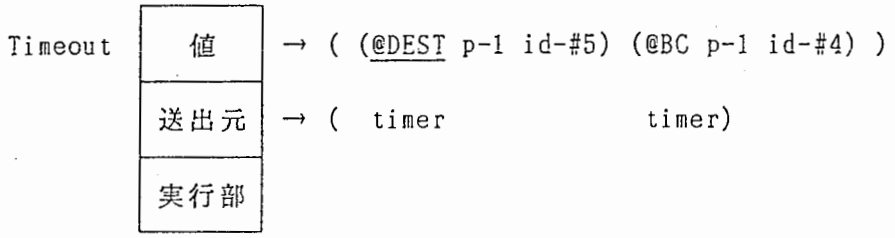
「実行idリスト」　：　(((p-1 id-#4) @BC))

(4)制約条件部での処理

①Timeoutイベント変数の値スロットの中に指定された「タイミングをとる相手」があれば条件は成立する。

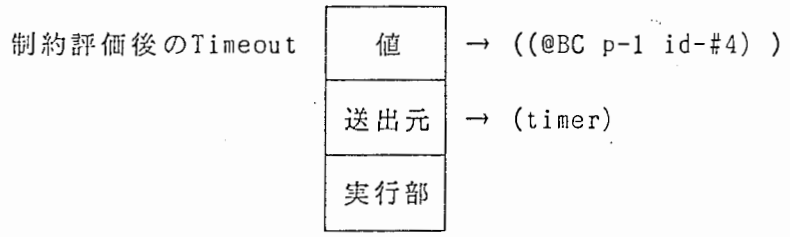
②評価後Timeoutイベント変数の送出元スロット、および値スロットから該当した要素だけを削除する。

（例）制約式：(((Timeout :@timer @DEST) ..) =〉 (( ) ..))



=〉

条件成立



8　ネットワーク

ネットワークコントロール部で表示系と処理系の間のデータのやりとりをする。

ネットワークコントロール部はネットワーク入力部とネットワーク出力部からなる。
①ネットワーク入力部 ： メッセージを受け取り、内部処理を行う。
②ネットワーク出力部 ： メッセージを送る。

なお、ネットワークを介してデータをやりとりする場合は、ストリングの形に変換して行う。

メッセージのハンドリングについて 図8.1 メッセージハンドリング に示す。

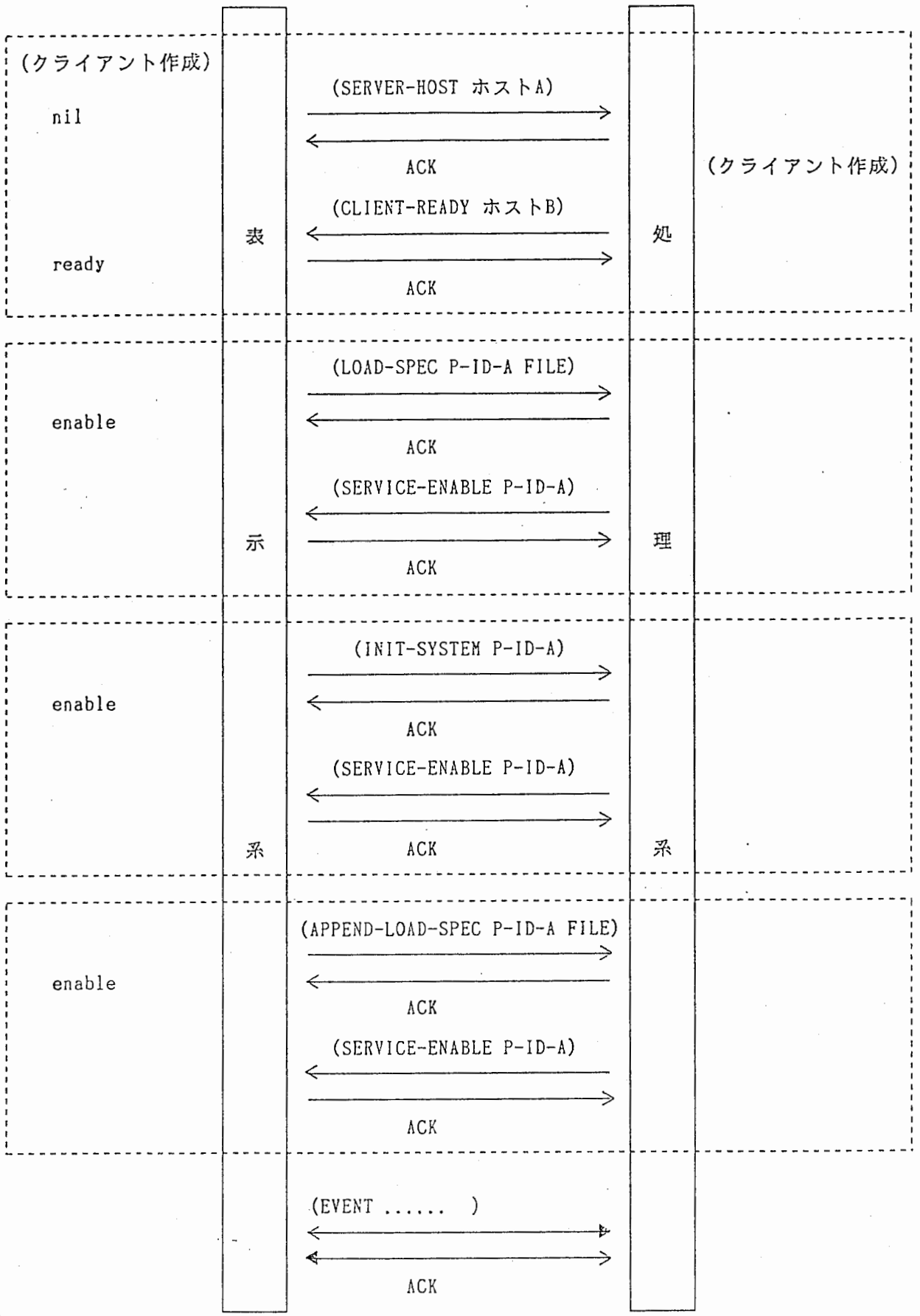


図8.1 メッセージハンドリング

8－1　ネットワーク入力部

処理系では、受け取ったものが、イベントコマンドであればＳ式に変換し入力バッファーに入れる。それ以外のコマンドであれば、それぞれのコマンドに該当する内部処理を行う。

表示系と処理系がやりとりするコマンドおよびそれに対応する処理は以下の通りである。

(1)処理系が表示系から受け取るコマンドおよび表示系が処理系から受け取るコマンド

◎Event

他プロセスからのイベントの受渡しを行う。
受け取った情報をＳ式にして、入力バッファーに入れる。

(2)表示系から処理系に送るコマンド

◎Server-Host

処理系のネットワーク出力部を起動する。
サービス状態を READY にする。
出力部を起動した後、表示系に CLIENT-READY コマンドを送出する。

◎Load-Spec

引数で指定された処理系対象仕様をロードし、処理系のイニシャライズ（変数および定数のインスタンスを作成する）を行う。
イニシャライズ終了後、サービス状態を ENABLE にし、表示系に SERVICE-ENABLE コマンドを送出する。
イニシャライズに失敗した場合（処理対象仕様に記述エラーがあった場合）は、READ-ERROR コマンドを表示系に送出する。

◎Init-System

処理系のイニシャライズ（変数および定数のインスタンスを作成する）を行う。
イニシャライズ終了後、サービス状態を ENABLE にし、表示系に SERVICE-ENABLE コマンドを送出する。
イニシャライズに失敗した場合は、READ-ERROR コマンドを表示系に送出する。

◎Append-Load-Spec

処理対象仕様を追加して処理系の再イニシャライズ（変数および定数のインスタンスを作成する）を行う。
イニシャライズ終了後、サービス状態を ENABLE にし、表示系に SERVICE-ENABLE コマンドを送出する。
イニシャライズに失敗した場合は、READ-ERROR コマンドを表示系に送出する。

(3)処理系から表示系に送るコマンド

◎Client-Ready

送ってきた処理系のサービス状態を READY にする。
折り返し、LOAD-SPEC コマンドを送出する。

◎Service-Enable

送ってきた処理系のサービス状態を ENABLE にする。

◎Read-Error

処理対象仕様の記述エラー情報が引数で渡される。

８－２　ネットワーク出力部

出力情報は、すべて出力バッファーに入ってくる。

ネットワーク出力部は、出力バッファーを調べ、情報が入っていたら１つ取りだし、出力情報をストリングに変換する。

バッファーに入ってくる出力情報には、先頭（ＣＡＲ部）に送出先へのストリーム情報が付与されているので、ネットワーク出力部では、ＣＡＲ部へストリングに変換した情報を送出する。

8－3　ネットワークコマンドフォーマット

表示系とやりとりするコマンドのフォーマットは以下の通りである。

(1)表示系から処理系へ送るコマンドおよび処理系から表示系へ送るコマンド

◎EVENT

::= (EVENT 〈 イベント変数 〉〈 送出先情報 〉〈 送出元情報 〉
〈 イベントｉｄ情報 〉〈 タイマー情報 〉〈 パラメーター 〉)

〈 イベント変数 〉
::= イベント変数名

〈 送出先情報 〉
::= 〈 送出先プロセスのid 〉〈 送出先プロセスのブロック 〉

〈 送出先プロセスのid 〉
::= イベントを受信するプロセスの識別子

〈 送出先プロセスのブロック 〉
::= イベントを受信するプロセスのブロック名
dcmp | func | supr | unknown

〈 送出元プロセスのid 〉
::= イベントを送信するプロセスの識別子

〈 送出元プロセスのブロック 〉
::= イベントを送信するプロセスのブロック名
dcmp | func | supr | unknown

〈 イベントｉｄ情報 〉
::= 〈 メッセージ-プロセスid 〉〈 メッセージ-シリアルナンバー 〉

〈 メッセージ-プロセスid 〉
::= 受け渡されるイベントに付与された識別子

〈 メッセージ-シリアルナンバー 〉
::= 受け渡されるイベントに付与されたシリアルナンバー

〈 パラメーター 〉
::= 受け渡されるイベントの値

(2)表示系から処理系に送るコマンド

◎SERVER-HOST

::= (SERVER-HOST 〈 ホスト名 〉)

〈 ホスト名 〉

::= 表示系のホスト名

◎LOAD-SPEC

::= (LOAD-SPEC 〈 プロセスid 〉〈 ファイルリスト 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

ここで指定されたものが、:@self の値となる。

〈 ファイルリスト 〉

::= (〈 ファイル名 〉*1)

〈 ファイル名 〉

::= 処理対象仕様のファイル名

◎INIT-SYSTEM

::= (INIT-SYSTEM 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

◎APPEND-LOAD-SPEC

::= (APPEND-LOAD-SPEC 〈 プロセスid 〉〈 ファイルリスト 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

ここで指定されたものが、:@self の値となる。

〈 ファイルリスト 〉

::= (〈 ファイル名 〉*1)

〈 ファイル名 〉

::= 処理対象仕様のファイル名

(3)処理系から表示系に送るコマンド

◎CLIENT-READY

::= (CLIENT-READY〈 ホスト名 〉)

〈 ホスト名 〉

::= 処理系のホスト名

◎SERVICE-ENABLE

::= (SERVICE-ENABLE〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

９　変数・定数インスタンス作成

９－１　ファイルの指定および読み込み

システムのイニシャライズ時に指定されたスペックファイルを読み込み、変数および定数のインスタンスを作成する。

スペックファイルは複数指定可能である。

スペックファイルの指定のタイミングは以下のいずれかである。

① 起動時に指定する。
　表示系が処理系にイニシャライズを行うコマンドを送出するときに引数で指定する。（「８－１　ネットワーク入力部」参照のこと）

② スペックファイルの記述の中に、取り込むファイルを指定する。
　スペックファイル中に追加スペック定義式を記述しておくことにより指定する。（「１０－５　スペックの追加」を参照のこと）

③ 起動後、新たにファイルを追加する。
　表示系が処理系にスペックファイルの追加コマンドを送出するときに引数で指定する。（「８－１　ネットワーク入力部」参照のこと）
　この場合、変数および定数のインスタンスは新たに作成し直すこととなる。

９－２　インスタンス作成

スペックに記述された変数定義式、定数定義式および制約定義式を解析し、簡単なシンタックスチェックを行い、変数および定数のインスタンスを作成する。

変数定義式を解析し、アドレス変数・状態変数のインスタンスを作成する。この時、指定されたＶＩＥＷが適切なものかをチェックする。
適切であれば、インスタンスのＶＩＥＷおよび値スロットにそれぞれセットする。

定数定義式を解析し、定数のインスタンスを作成する。この時、指定されたＶＩＥＷが適切なものかをチェックする。
適切であれば、インスタンスのＶＩＥＷおよび値スロットにそれぞれセットする。

制約定義式を解析し、イベント条件式で使用されているイベント変数のインスタンスを作成する。また実行部スロットにこの変数が、分配部・機能部・同期部のどの条件部で使われているかをセットする。
この時点では値は不定である。

また、制約定義式を解析し、一時変数のインスタンスを作成する。

１０　言語仕様

１０－１　変数および定数

１０－１－１　変数の種類

変数として以下の４種類を用いる。

１）イベント変数
２）アドレス変数
３）状態変数
４）一時変数

１０－１－２　イベント変数

イベント変数は各サービスプロセス間（または１つのサービスプロセス内）での情報の流れを表すもので、以下のように記述する。

〈 英字 〉〈 英数字 〉

ただし”Timeout”イベント（他のプロセスへイベントを送出し、その応答が所定時間内に返ってこなかった場合に生じるイベント）は、予約語的な扱いを受ける。

１０－１－３　アドレス変数

アドレス変数は、あるプロセスへイベントを送出する場合の送り先、または他のプロセスからイベントを受け取る場合のその送り元を特定するための変数である。
アドレス変数は以下のように記述する。

@〈 英数字列 〉

アドレス変数として以下の予約語（定数）がある。

```
:@self     : 自分のプロセス
:@dcmp     : 自プロセス内の分配部
:@func     : 自プロセス内の機能部
:@supr     : 自プロセス内の同期部
:@device   : 自プロセスの扱う端末ないしＩ／Ｏ
:@timer    : タイマー
```

１０－１－４　状態変数

状態変数はプロセス内の現時点での状態を制御するために用いる。
状態変数は以下のように記述する。

$〈 英数字列 〉

１０－１－５　一時変数

一時変数は、制約式内で一時的に値をバインドしておくために用いる。制約式評価中にこの一時変数に値がバインドされた場合に、その制約式中の同一の一時変数はこのバインドされた値に置き換えられて制約式が評価される。
一時変数は以下のように記述する。

*〈 英数字列 〉

また一時変数は一制約式中でのみ有効である。

１０－１－６　変数の宣言

制約式中のアドレス変数、状態変数は制約式とは別にあらかじめ宣言しておかなければならない。
変数は以下のように宣言する。

```
( setvar 〈 VIEW 〉〈 変数名 〉〈 初期値 〉 )
```

ただし、〈 変数名 〉〈 初期値 〉の組は任意個繰り返してもよい。
また〈 VIEW 〉は、「１０－１－９　変数・定数のＶＩＥＷ」の項を参照のこと。

１０－１－７　定数

制約式には値（数字、シンボル etc.）を直接記述することもできるが、この値をあらかじめ変数にバインドしておき、制約式中ではこの変数を定数として使うことが可能である。
定数は以下のように記述する。

```
%〈 英数字列 〉
%@〈 英数字列 〉
```

ここで、%@〈 英数字列 〉型の定数を「アドレス定数」と呼ぶ。
「１０－１－３　アドレス変数」中の予約語はアドレス定数である。

１０－１－８　定数の宣言

定数は制約式とは別にあらかじめ宣言しておかなければならない。
定数は以下のように宣言する。

```
（ setconst 〈 VIEW 〉〈 定数名 〉〈 値 〉）
```

ただし、〈 定数名 〉〈 初期値 〉の組は任意個繰り返してもよい。
また〈 VIEW 〉は、「１０－１－９　変数・定数のＶＩＥＷ」の項を参照のこと。

１０－１－９　変数・定数のＶＩＥＷ

宣言される変数、定数はその値の有効範囲を指定しなければならない。この変数、定数の有効範囲を ＶＩＥＷ という。

変数、定数のＶＩＥＷは以下の通りである。

:global	：プロセス内で有効
:interactive	：プロセス内の分配部、同期部でのみ有効
:decomposing	：プロセス内の分配部でのみ有効
:superposing	：プロセス内の同期部でのみ有効
:functional	：プロセス内の機能部でのみ有効

またこのＶＩＥＷには包含関係があり、宣言されたＶＩＥＷに包含関係がない場合には両方のＶＩＥＷで別々の値を取ることが出来る。

ＶＩＥＷの包含関係は以下の通りである。

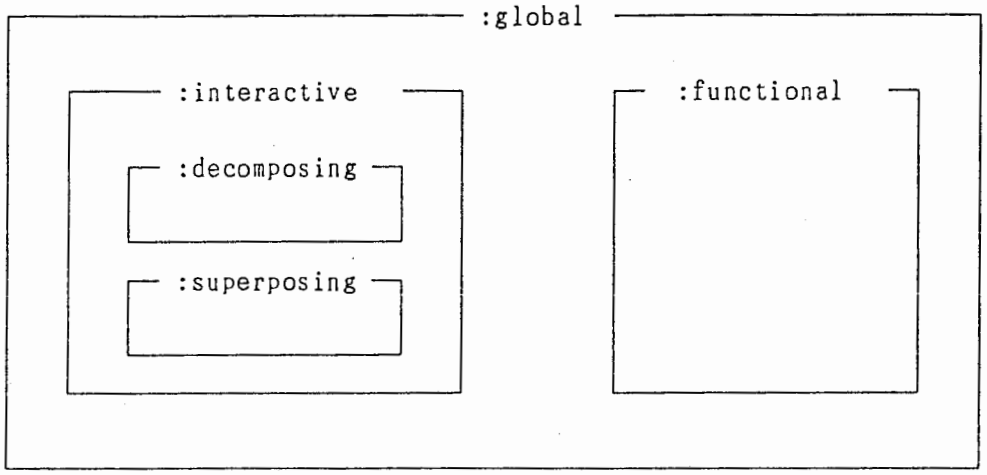


図10.1 変数・定数のスコープ

１０－２　制約式

制約式は以下のように記述する。

```
（（〈 条件部 〉）=〉（〈 ボディ部 〉））
```

１０－２－１　制約式の評価

条件部は複数の条件式を記述することができる。このとき各条件式はＡＮＤ条件で結ばれているとみなす。

また、条件式としてＯＲ条件の条件式が記述できる。

ＯＲ条件中の条件式にＡＮＤ条件の条件式を記述する場合にはＡＮＤ条件を陽に指定するか、ＡＮＤ条件となる条件式を（）で囲む。

ＡＮＤ、ＯＲの条件は、何重のネスティングも許す。

１０－２－２　条件式の種類

条件式には以下のものがある。

1）イベント条件式
2）変数条件式
3）ＡＮＤ条件式
4）ＯＲ条件式

１０－２－３　イベント条件式

イベント変数の送出元、値の判定を行う。

イベント条件式は以下のように記述する。

```
（〈 イベント変数 〉〈 送出元 〉〔〈 パラメータリスト 〉〕）
```

１０－２－４　変数条件式

変数（イベント変数以外）の値の判定を行う。
変数条件式は以下のように記述する。

```
（｛〈 条件否定 〉｝〈 変数名 〉〈 パラメータリスト 〉）
```

ここで〈 条件否定 〉が指定されていた場合、〈 変数 〉のもつ値が〈 値リスト 〉と等しければ変数条件式を満足しないものとし、そうでなければ変数条件式が満足する。

１０－２－５　ＡＮＤ条件式

条件部の全条件式が満たされれば条件は成立する。
ＡＮＤ条件は以下のように記述する。

```
（ AND〈 条件式 〉）
（〈 条件式 〉）
〈 条件式 〉
```

１０－２－６　ＯＲ条件式

条件部の条件式が１つでも満たされていれば条件は成立する。
ただし、ＯＲ条件式の評価は、「ＯＲ条件式中の全条件式を１回評価し、そのうち１つでも条件が成立していればＯＲ条件は成立する」というやり方で行う。
ＯＲ条件式は以下のように記述する。

```
（ OR〈 条件式 〉）
OR〈 条件式 〉
```

この時、下のＯＲ条件式は条件式がＯＲ条件式のみの場合に許される記述であり、それ以外は条件式の記述エラーである。

１０－２－７　ボディ部の評価

ボディ部には複数のボディ式を記述することができる。制約式評価中、条件部の全ての条件式が満足された場合にボディ部が評価される。ボディ部の評価はそこに記述されている全ボディ式を順に評価する。

１０－２－８　ボディ式の種類

ボディ式には以下のものがある。

1）イベント定義式
2）状態定義式
3）タイマー解除式

１０－２－９　イベント定義式

イベントをあるプロセスへ送る。
イベント定義式は以下のように記述する。

```
（〈 イベント変数名 〉〈 送出先 〉〔〈 パラメータリスト 〉〕
                                〔〈 タイマー設定式 〉〕　）
```

イベントを他のプロセスへ送る場合、その応答がある時間内になければならない場合、その時間を〈 タイマー設定式 〉を用いて設定することができる。
〈 タイマー設定式 〉は以下のように記述する。

```
:Timing〈 タイマー値 〉
```

また〈タイマー値〉の単位は１００ｍｓである。

１０－２－１０　状態定義式

状態変数へ値を代入する。
状態定義式は以下のように記述する。

```
（〈 状態変数 〉〈 値 〉）
```

１０－２－１１　タイマー解除式

イベント送出時に設定されたタイマーを陽に解除する。
タイマー解除式は以下のように記述する。

```
（ Releasetiming :@timer〈 タイミング相手 〉）
```

タイマーの解除は、イベント送出時に設定された時間内に送出先よりイベントが返ってくれば自動的に行われるが、タイマー解除式を使って陽に行うこともできる。

１０－３　制約定義式

制約式は各処理単位毎にまとめて以下のように記述する。

```
( defcontstraint 〈 タイプ 〉〈 制約式 〉 )
```

１０－３－１　制約定義式のタイプ

制約式は、各処理単位（分配部、機能部、同期部）毎にまとめて上記のように記述される。この各処理単位の指定を〈 タイプ 〉で行う。
制約式のタイプには以下のものがある。

:decomposing　：分配部
:functional　：機能部
:superposing　：同期部

制約式を記述する際に、例えば分配部の制約定義式を複数記述しても構わない。

１０－４　制約の評価

１０－４－１　分配部

分配部は入力されたイベントを自プロセスの機能部を関連する外部プロセスへ分配するためのものである。

分配部中の制約式の評価は、値がバインドされているイベント変数と状態変数の値により、条件部が真になる制約式のボディ部を評価することにより行われる。この時、制約式で評価したイベント変数の値は不定に戻す。

分配部では、先頭から条件式が真になる分配制約をチェックし、全分配制約の条件式のチェックが終了した後、条件部が真になる制約式のボディ部を評価する。

評価は全制約に対して１通り１度だけ実行し、制御を機能部に移す。

１０－４－２　機能部

機能部は、プロセス個別の機能に関する動作の記述を行う。

機能部中の制約式の評価は、分配部と同様に行われる。また制約式で評価したイベント変数の値は不定に戻される。

機能部も分配部と同様、先頭から条件式が真になる機能制約をチェックし、全機能制約の条件式のチェックが終了した後、条件部が真になる制約式のボディ部を評価する。

評価は全制約に対して１通り、１度だけ実行し制御を同期部に移す。

１０－４－３　同期部

同期部は自プロセスの動作が外部プロセスの状態／出力と同期を取って決まるような場合の制約を記述する。

同期部中の制約式の評価は、基本的には分配部、機能部の制約式の評価と同様である。

同期部では、先頭から条件式が真になる同期制約がチェックされ、条件式が真になればその時点でボディ部が評価される。また制約式で評価したイベント変数の値は条件式が真になった値のみが除かれる。この時値が何もなければイベント変数の値を不定に戻す。

評価は再帰的に何度でも行われ、条件式に現れる値のバインドされているイベント変数がなくなれば、つまり条件式中の全イベント変数の値が不定になれば制御を分配部または機能部に移す。

同期制約の評価が終了した後、制御は分配部へ移るのが基本であるが、同期制約評価中に自プロセスの機能部へイベントを送出した場合は、制御は機能部へ移る。

１０－５ ＣＳＬの追加

ＣＳＬの中で、別ファイルで定義したＣＳＬを明示的に取り込むことができる。
この場合、変数定義式、定数定義式はそれぞれ１つにまとめて処理される。
制約定義式は、分配部・機能部・同期部の各実行部ごとにまとめて処理される。

ＣＳＬの追加は以下のように宣言する。

```
( include-constraint 〈 ＣＳＬファイル名 〉 )
```

１０－６　言語記述

以下に制約式のフォーマットを ＢＮＦ で示す。

〈 ＣＳＬ 〉
　　::= 〈 変数定義式 〉*1 〈 定数定義式 〉*1 〈 制約定義式 〉
　　　　〈 追加ＣＳＬ定義式 〉*1 〈 ＣＳＬ 〉*1

〈 変数定義式 〉
　　::= (setvar 〈 VIEW 〉 〈 変数式 〉*2)

〈 VIEW 〉
　　::= :global | :interactive | :functional | :decomposing | :superposing

〈 変数式 〉
　　::= 〈 変数 〉〈 値 〉

〈 変数 〉
　　::= 〈 状態変数 〉 | 〈 アドレス変数 〉

〈 状態変数 〉
　　::= $〈 英数字列 〉

〈 アドレス変数 〉
　　::= @〈 英数字列 〉

〈 値 〉
　　::= 〈 数値 〉 | 〈 文字列 〉 | 〈 シンボル 〉 | 〈 リスト 〉

〈 定数定義式 〉
　　::= 〈 setconst 〈 VIEW 〉〈 定数式 〉*2)

〈 定数式 〉
　　::= 〈 定数 〉〈 値 〉

〈 定数 〉
　　::= %〈 英数字列 〉 | 〈 アドレス定数 〉

〈 アドレス定数 〉
　　::= %@〈 英数字列 〉 | :@self | :@dcmp | :@func | :@supr |
　　　　:@timer | :@device

〈 制約定義式 〉
　　::= (defconstraint 〈 タイプ 〉〈 制約式 〉*2)

〈 タイプ 〉
　　::= :decomposing | :functional | :superposing

〈 制約式 〉

::= ((〈 条件式 〉) =〉 (〈 ボディ式 〉*2))

〈 条件式 〉
::= 〈 イベント条件式 〉 | 〈 変数条件式 〉 |
〈 ＡＮＤ条件式 〉 | 〈 ＯＲ条件式 〉

〈 イベント条件式 〉
::= (〈 イベント変数 〉〈 送出元 〉〈 パラメータリスト 〉*1)

〈 イベント変数 〉
::= 〈 英字 〉{ 〈 英数字列 〉 }

〈 送出元 〉
::= 〈 アドレス変数 〉 | 〈 アドレス定数 〉

〈 パラメータリスト 〉
::= 〈 変数〉 | 〈 一時変数 〉 | 〈 イベント変数 〉 | 〈 定数 〉 | 〈 値 〉

〈 一時変数 〉
::= *〈 英数字列 〉

〈 変数条件式 〉
::= ({ 〈 条件否定 〉 } 〈 変数 〉〈 パラメータリスト 〉*1)

〈 条件否定 〉
::= ~

〈 ＡＮＤ条件式 〉
::= (and 〈 条件式 〉*2) | (〈 条件式 〉*2) | 〈 条件式 〉*2

〈 ＯＲ条件式 〉
::= (or 〈 条件式 〉*2) | or 〈 条件式 〉*2

〈 ボディ式 〉
::= 〈 イベント定義式 〉 | 〈 状態定義式 〉 | 〈 タイマー解除式 〉

〈 イベント定義式 〉
::= (〈 イベント変数 〉〈 送出先 〉〈 パラメータリスト 〉*1
{ 〈 タイマー設定式 〉 })

〈 送出先 〉
::= 〈 アドレス変数 〉 | 〈 アドレス定数 〉

〈 タイマー設定式 〉
::= :timing 〈 タイマー時間 〉

〈 タイマー時間 〉
::= 数値 | 値が数値となっている定数 | 〈 タイマー変数 〉 （単位：100ms）

〈 タイマー変数 〉
::= 値が数値となっている変数

〈 状態定義式 〉
::= （ 〈 状態変数 〉 〈 パラメータリスト 〉*1 ）

〈 タイマー解除式 〉
::= （ Releasetiming :@timer 〈 タイミング相手 〉 ）

〈 タイミング相手 〉
::= 〈 アドレス変数 〉 | 〈 アドレス定数 〉

〈 追加ＣＳＬ定義式 〉
::= （ include-constraint 〈 ＣＳＬファイル 名 〉 ）

〈 ＣＳＬファイル名 〉
::= ＣＳＬ記述ファイル名

*1 0個以上任意個
*2 1個以上任意個
{} 存在任意

付録１

ネットワーク仕様処理系

デバッグ環境およびネットワークコマンド説明書

１９９０年８月

目次

１．　概要

平成元年度作成したシステムにデバッグ環境を追加する。
これは表示系と処理系の間での内部コマンドのやりとりによって、以下の機能を実現するものである。

①処理系の動作環境を設定する
②処理系の情報を参照する
③処理系の変数の値を変更設定する

内部コマンドのやりとりは、既に実現しているネットワークコントロール部を経由してのコマンドのやりとりと同様の方法で行う。
すなわち、ネットワーク入力部がコマンドを受け取り適当な内部処理を行う。
外部にコマンドを送出する場合は、コマンドを出力バッファーにいれ、ネットワーク出力部がそこからコマンドを取りだし適切なプロセスに送る。
ただし処理系のデバッグ環境の出力コマンドは、出力情報のシーケンスを保つため、一旦イベントコマンドと同じ出力キューに入れ、出力部（ネットワーク出力部とは別）が出力キューから取りだして出力バッファーに入れる。
起動時に表示系に送出するコマンドについては直接出力バッファーに入れる。

本説明書では、表示系と処理系の間で受け渡されるコマンドについて、既に実現しているコマンドもあわせて説明する。

２．　ネットワークコントロール部機能概要

ネットワークコントロール部はネットワーク入力部とネットワーク出力部からなる。
①ネットワーク入力部　：　メッセージを受け取り、内部処理を行う。
②ネットワーク出力部　：　メッセージを送る。

なお、ネットワークを介してデータをやりとりする場合は、ストリングの形に変換して行う。

内部コマンドの流れを「図3.1 内部コマンドの流れ」に示す。

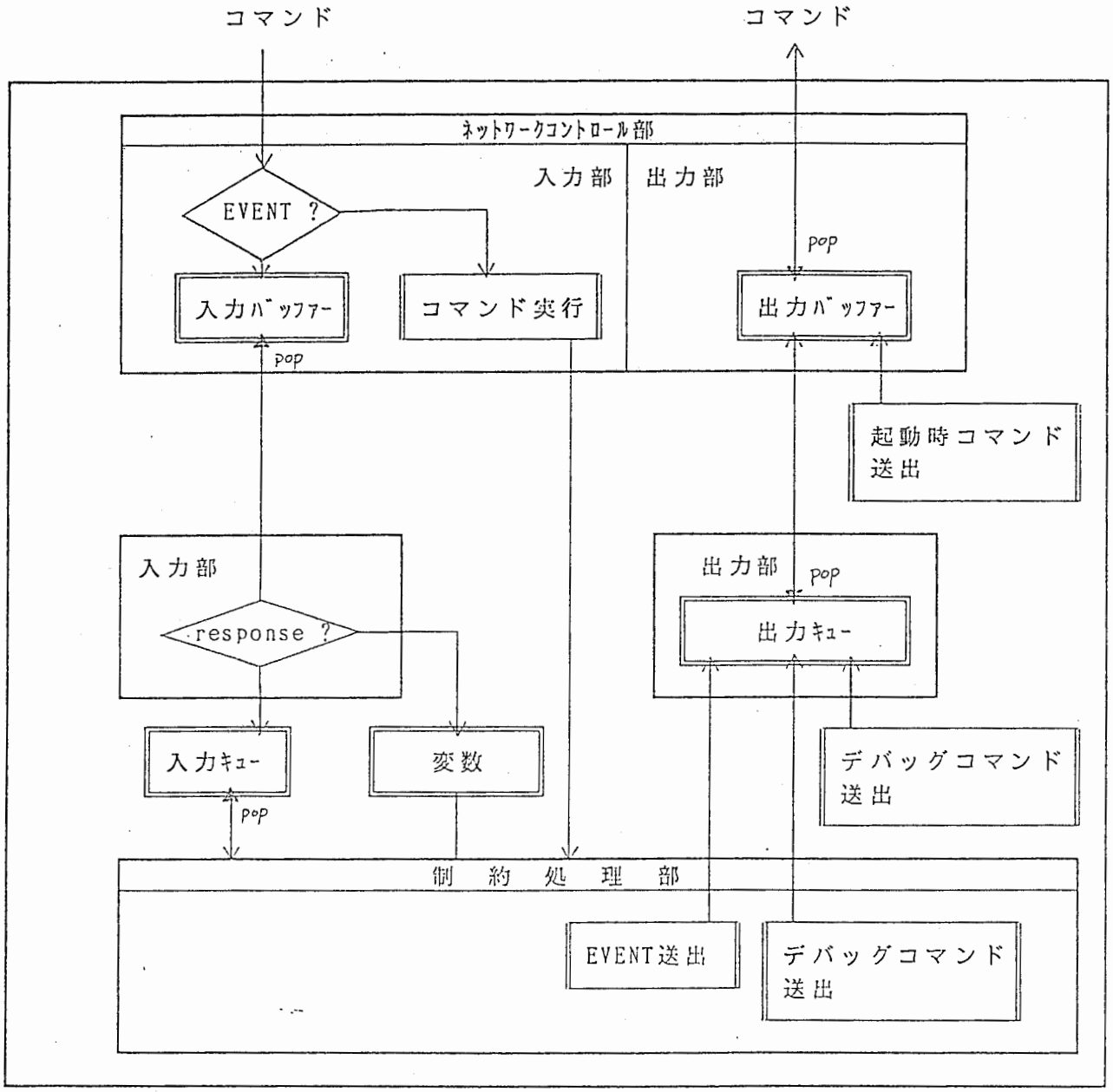


図3.1 内部コマンドの流れ

３．実行コマンド

３．１　コマンド一覧

表示系と処理系がやりとりするコマンドは以下の通りである。

(1)表示系から処理系に送るコマンド

1. Append-Load-Spec
2. Init-System
3. Invoke-Next-Event-Transaction
4. Load-Spec
5. Resume-Event-Transaction
6. Server-Host
7. Set-Mode
8. Set-Variables-Value
9. Show-Constraints
10. Show-Exec-part
11. Show-Variables-Value
12. Suspend-Event-Transaction

(2)処理系から表示系に送るコマンド

1. Client-Ready
2. Constraints
3. Current-Exec-Part
4. Fired-Rule
5. Read-Error
6. Result-Set-Variables-Value
7. Service-Enable
8. Variables-Value

（注）　処理系の起動時に表示系に送るコマンド(1. 5. 7.)は直接出力バッファーに入れる。
それ以外のコマンドについては一旦出力キューに入れ、出力部（ネットワーク出力部とは別）が出力キューから取りだし、出力バッファーに入れる。

(3)イベント

1. Event

3．2　コマンドの処理

(1)表示系から処理系に送るコマンドとそれを受け取った場合の処理系の処理内容

1.Append-Load-Spec

（処理）

処理対象仕様を追加して読み込み、処理系の再イニシャライズ（変数および定数のインスタンスを作成する）を行う。

イニシャライズ終了後、サービス状態を ENABLE にし、表示系に SERVICE-ENABLE コマンドを送出する。

イニシャライズに失敗した場合は、表示系に READ-ERROR コマンドを送出する。

2.Init-System

（処理）

処理系のイニシャライズ（変数および定数のインスタンスを作成する）を行う。

イニシャライズ終了後、サービス状態を ENABLE にし、表示系に SERVICE-ENABLE コマンドを送出する。

イニシャライズに失敗した場合は、表示系に READ-ERROR コマンドを送出する。

3.Invoke-Next-Event-Transaction

（処理）

サスペンド状態において（12.Suspend-Event-Transaction コマンドによってこの状態になる）、１つの入力イベント（外部プロセスおよび device からのイベント）の処理を行う。

実行している部が分配部の場合は、入力キューにイベントが入っていればそこからイベントを１つ取りだし値をセットし分配部の制約チェックを開始する。入っていなければ入力バッファーからイベントを取り出しキューイングする処理を行う。

実行している部が分配部以外の場合は、入力バッファーから自プロセスへの返答のイベントが見つかるか（見つかったら値をセットする）または空になるまでイベントを取り出す。この場合、キューイングするイベントがあればキューイングする。

4.Load-Spec

（処理）

処理系対象仕様を読み込み、処理系のイニシャライズ（変数および定数のインスタンスを作成する）を行う。

イニシャライズ終了後、サービス状態を ENABLE にし、表示系に SERVICE-ENABLE コマンドを送出する。

イニシャライズに失敗した場合（処理対象仕様に記述エラーがあった場合）は、表示系に READ-ERROR コマンドを送出する。

5. Resume-Event-Transaction

（処理）

サスペンド状態を解除する。

6. Server-Host

（処理）

処理系のネットワーク出力部を起動する。

サービス状態を READY にする。

出力部を起動した後、表示系に CLIENT-READY コマンドを送出する。

7. Set-Mode

（処理）

処理系の実行モードを指定されたモード（デバッグモードまたはノーマルモード）に変更する。

初期状態はノーマルモードである。

デバッグモードにおいては、実行している部が移った場合 CURRENT-EXEC-PART コマンドを表示系に送る。

また、ある制約式の条件部が成立すると FIRED-RULE コマンドを表示系に送る。

8. Set-Variables-Value

（処理）

指定された変数について指定された VIEW の値を変更する。

このコマンドを実行した結果、RESULT-SET-VATIABLES-VALUE コマンドを以下の情報と共に表示系に送る。

① 変数をセットした場合、その変数名と値をセットした VIEW および値。

② 指定された変数がない場合または指定した VIEW を間違えていた場合は、変数名と値をセットしようとした VIEW およびエラー情報。

9. Show-Constraints

（処理）

指定された実行部の制約式を表示系に送出する。

10. Show-Exec-part

（処理）

現在実行している部（分配部の場合は入力待状態か制約式をチェックしているかの区別をつける）を表示系に送出する。

11. Show-Variables-Value

（処理）

setvar で変数定義された全ての変数について変数名と VIEW およびその値を表示系に送出する。

12. Suspend-Event-Transaction

（処理）

処理系をサスペンド状態（外部からのイベントの読み込みを中止するため、処理が中断される状態）にする。

(2)処理系から表示系に送るコマンドとその情報または受け取った場合の処理

1.Client-Ready
（処理）
送ってきた処理系のサービス状態を READY にする。
折り返し、LOAD-SPEC コマンドを送出する。

2.Constraints
（情報）
SHOW-CONSTRAINTS の結果送られてきた制約式。

3.Current-Exec-Part
（情報）
処理系が実行している部。

4.Fired-Rule
（情報）
条件部が成立した制約式。
および、制約式の中で一時変数が使用されていた場合は、その一時変数と値の情報。

5.Read-Error
（情報）
処理対象仕様の記述エラー情報。

6.Result-Set-Variables-Value
（情報）
SET-VARIABLES-VALUE コマンドの実行結果。
変数をセットした場合、その変数名と値をセットした VIEW および値（リスト）のリスト。
指定された変数がない場合または指定した VIEW を間違えていた場合は、変数名と値をセットしようとした VIEW およびエラー情報（ストリング）のリスト。

7.Service-Enable
（処理）
送ってきた処理系のサービス状態を ENABLE にする。

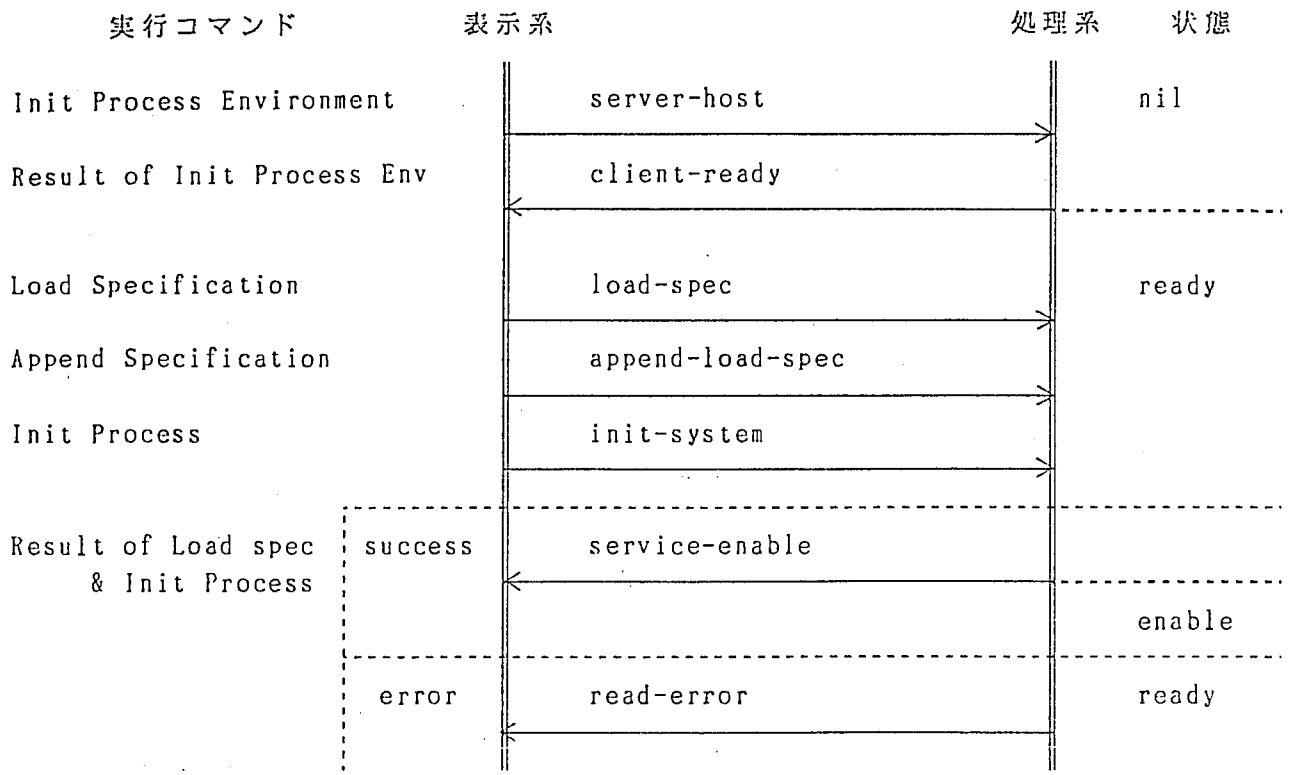
8.Variables-Value
（情報）
処理対象仕様の中で setvar で定義した変数と VIEW およびその値。

(3)イベント

1.Event
（処理）
他プロセスからのイベントの受渡しを行う。
受け取った情報をＳ式にして、入力バッファーに入れる。

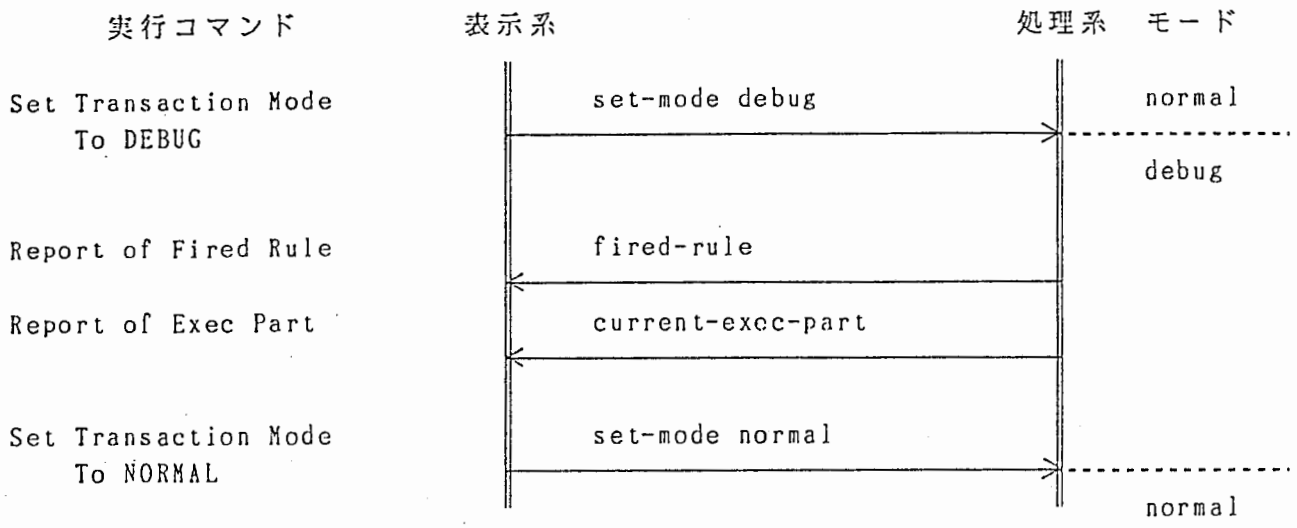
４．コマンドシーケンス

４．１　処理系起動時



4．2　デバッグモード実行時

処理系の状態が enable の時のみ有効。

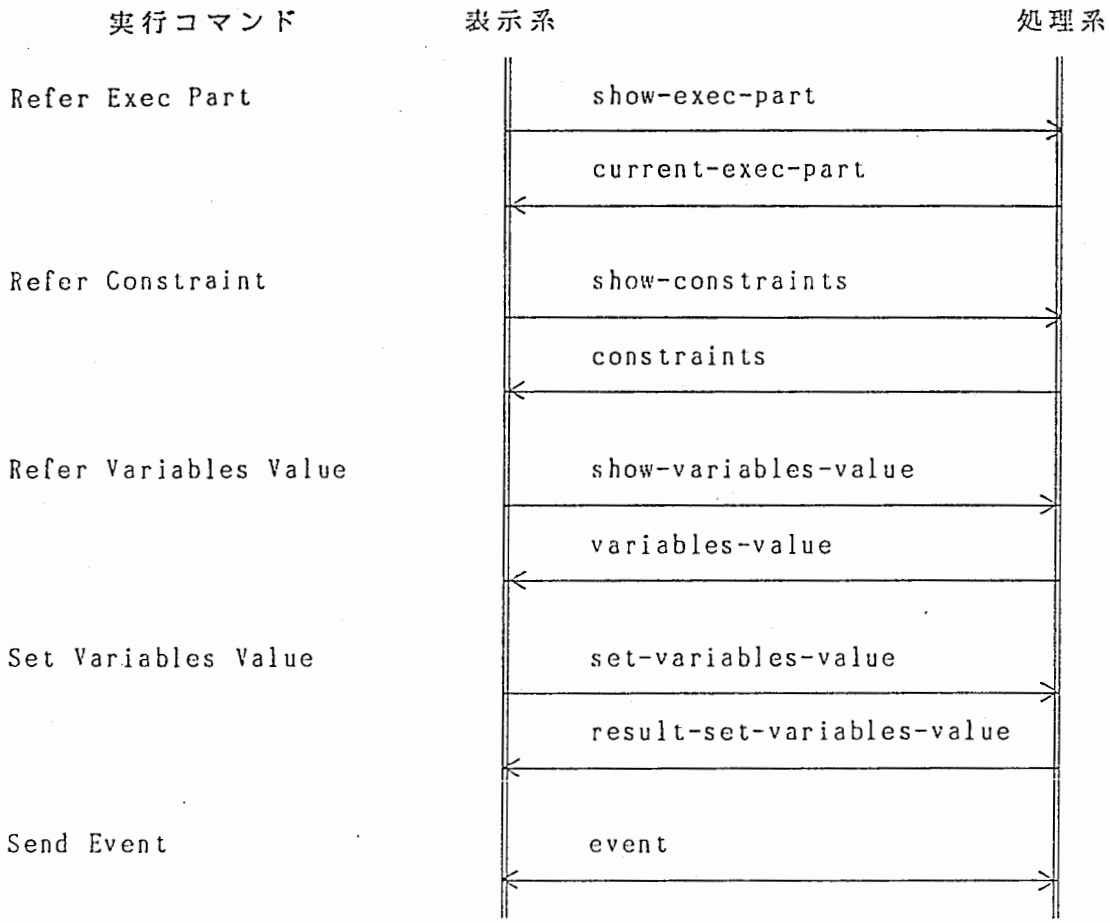


4．3　サスペンド実行

処理系の状態が enable の時のみ有効。

4．4　その他

処理系の状態が enable の時のみ有効。



５．　コマンドフォーマット

表示系と処理系の間で受け渡される内部コマンドのフォーマット（入力バッファーに入れるコマンド）は以下の通りである。

(1)表示系から処理系に送るコマンド

1.Append-Load-Spec

::= (APPEND-LOAD-SPEC 〈 プロセスid 〉 (〈 ファイル名 〉*1))

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 ファイル名 〉

::= 処理対象仕様のファイル名

2.Init-System

::= (INIT-SYSTEM 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

3.Invoke-Next-Event-Transaction

::= (INVOKE-NEXT-EVENT-TRANSACTION 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

4.Load-Spec

::= (LOAD-SPEC 〈 プロセスid 〉 (〈 ファイル名 〉*1))

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号
ここで指定されたものが、:@self の値となる。

〈 ファイル名 〉

::= 処理対象仕様のファイル名

5.Resume-Event-Transaction

::= (RESUME-EVENT-TRANSACTION 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

6.Server-Host

::= (SERVER-HOST 〈 ホスト名 〉)

〈 ホスト名 〉

::= 表示系のホスト名

7.Set-Mode

::= (SET-MODE 〈 プロセスid 〉〈 モード 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 モード 〉

::= normal | debug

8.Set-Variables-Value

::= (Set-VARIABLES-VALUE 〈 プロセスid 〉 (〈 変数リスト 〉*1))

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 変数リスト 〉

::= (〈 変数 〉〈 ＶＩＥＷ 〉 値)

〈 変数 〉

::= 状態変数 | アドレス変数

〈 ＶＩＥＷ 〉

::= global | decomposing | functional | superposing
| interactive

9.Show-Constraints

::= (SHOW-CONSTRAINTS 〈 プロセスid 〉〈 実行部 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 実行部 〉

::= decomposing | functional | superposing

10.Show-Exec-part

::= (SHOW-EXEC-PART 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

11.Show-Variables-Value

::= (SHOW-VARIABLES-VALUE 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

12.Suspend-Event-Transaction

::= (SUSPEND-EVENT-TRANSACTION 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

(2)処理系から表示系に送るコマンド

1.Client-Ready

::= (CLIENT-READY 〈 ホスト名 〉)

〈 ホスト名 〉

::= 処理系のホスト名

2.Constraints

::= (CONSTRAINTS 〈 プロセスid 〉〈 実行部 〉(制約式*1)))

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 実行部 〉

::= decomposing | functional | superposing

3.Current-Exec-Part

::= (CURRENT-EXEC-PART 〈 プロセスid 〉〈 実行部 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 実行部 〉

::= idling | decomposing | functional | superposing

4.Fired-Rule

::= (FIRED-RULE 〈 プロセスid 〉 制約式 〈 一時変数リスト 〉*2)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 一時変数リスト 〉

::= ((一時変数 値)*1)

5.Read-Error

::= (READ-ERROR 〈 プロセスid 〉(〈 ファイル名 〉*1)
エラー情報)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 ファイル名 〉

::= 処理対象仕様のファイル名

6.Result-Set-Variables-Value

::= (Result-Set-Variables-Value 〈 プロセスid 〉〈 変数リスト 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 変数リスト 〉

::= (〈 変数 〉〈 VIEW 〉〈 実行結果 〉)

〈 実行結果 〉

::= 値(リスト) | 〈 エラー情報 〉

〈 エラー情報 〉

::= "ERROR エラーの内容 "(ストリング)

7.Service-Enable

::= (SERVICE-ENABLE 〈 プロセスid 〉)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

8.Variables-Value

::= (VARIABLES-VALUE 〈 プロセスid 〉 (〈 変数リスト 〉*1)

〈 プロセスid 〉

::= サービスプロセス名または基本呼番号

〈 変数リスト 〉

::= (〈 変数 〉〈 ＶＩＥＷ 〉値)

〈 変数 〉

::= 状態変数 | アドレス変数

〈 ＶＩＥＷ 〉

::= global | decomposing | functional | superposing
| interactive

(3)イベント

1.Event

::= (EVENT 〈 イベント変数 〉〈 送出先情報 〉〈 送出元情報 〉
〈 イベントｉｄ情報 〉〈 タイマー情報 〉〈 パラメーター 〉)

〈 イベント変数 〉

::= イベント変数名

〈 送出先情報 〉

::=〈 送出先プロセスのid 〉〈 送出先プロセスのブロック 〉

〈 送出先プロセスのid 〉

::= イベントを受信するプロセスの識別子

〈 送出先プロセスのブロック 〉

::= イベントを受信するプロセスのブロック名
dcmp | func | supr | unknown

〈 送出元プロセスのid 〉

::= イベントを送信するプロセスの識別子

〈 送出元プロセスのブロック 〉

::= イベントを送信するプロセスのブロック名
dcmp | func | supr | unknown

〈 イベントｉｄ情報 〉

::=〈 メッセージ-プロセスid 〉〈 メッセージ-シリアルナンバー 〉

〈 メッセージ-プロセスid 〉

::= 受け渡されるイベントに付与された識別子

〈 メッセージ-シリアルナンバー 〉

::= 受け渡されるイベントに付与されたシリアルナンバー

〈 パラメーター 〉

::= 受け渡されるイベントの値

*1 1個以上任意個

*2 存在任意

６．　ファイル構成

ここでは、デバッグ環境の実現およびネットワークコマンドのやりとりに関連するディレクトリおよびファイル構成について説明する。

関連するディレクトリは以下の通りである。

①communication
②network

6．1　communication ディレクトリ

このディレクトリには、入力部および出力部の実行に関するファイルが含まれている。以下のファイルがある。

①global-variables.lisp
②in-process.lisp
③out-process.lisp
④utility.lisp

①global-variables.lisp

入力部および出力部で使用するグローバル変数を定義する。

②in-process.lisp

入力部の処理を行う関数を定義する。

③out-process.lisp

出力部の処理を行う関数を定義する。

④utility.lisp

入力部および出力部で使用するユーティリティ関数を定義する。

6．2　network ディレクトリ

このディレクトリには、ネットワークコントロール部（入力部および出力部）およびデバッグ環境の実行に関するファイルが含まれている。
以下のファイルがある。

①flavor-definition.lisp
②global-variables.lisp
③network-io.lisp
④load-configuration.lisp
⑤debugger.lisp
⑥utility.lisp

①flavor-definition.lisp

ネットワークコントロール部で使用するフレーバを定義するファイルである。

②global-variables.lisp

ネットワークコントロール部で使用するグローバル変数を定義する。

③network-io.lisp

ネットワークコントロール部の処理を行う関数を定義する。
プロトコル定義およびサーバー定義も行う。

④load-configuration.lisp

コンフィギュレーションファイルを読み込んで処理系のコンフィギュレーションを管理するテーブルを作成する関数を定義する。

⑤debugger.lisp

デバッグコマンドを受け取ったあとの実際の処理を行う関数を定義する。

⑥utility.lisp

ネットワークコントロール部で使用するユーティリティ関数を定義する。

7. グローバル変数

ここでは、デバッグ環境の実現およびネットワークコマンドのやりとりに関連する、グローバル変数および名前つき定数を凡例に従って説明する。

（凡例）

変数名 〔特殊形式名〕*1 （とりうる値のタイプ）

*1 defparameter で定義された変数 : parameter
defconstant で定義された名前付き定数: constant
記述のないものは defvar で定義された変数である。

7. 1 communication ディレクトリ

・*input-buffer* （リスト）

ネットワークコントロール入力部で、表示系から送られてきたイベント（eventコマンド）を入れる（最後に追加）。
入力部がここから要素を取り出して、入力キューに追加、値のセットあるいはそのイベントを捨てる処理を行う。

・*output-buffer* （リスト）

表示系に送出するコマンドを入れる（最後の追加）。
処理系起動時のコマンドはネットワークコントロール部で直接追加する。
それ以外のコマンドについては、出力部が出力キューに入っているコマンドを取り出して、表示系のストリームをコマンドの先頭に追加して、この変数に入れる。
ネットワークコントロール出力部がここから要素を取り出して、ストリングに変換し、先頭のストリームに情報を送る。

・*input-que* （リスト）

外部プロセスからのイベントを入れる（最後に追加）。
入力部がメッセージidから、当該処理系に対する新たなイベントと判断したものについてここに入れる。
制約処理部の制御が分配部に移った時点でここから要素を１つ取りだして値をセットし、分配制約の評価を開始する。

・*output-que* （リスト）

制約処理部から表示系に送るイベント情報および、表示系から送られてきたデバッグ用コマンドへの返答や実行モードに応じて表示系に送るデバッグコマンド情報を入れる（最後に追加）。
出力部がここから要素を取り出して出力バッファーに追加する。

7．2　network ディレクトリ

・*host-name*　（シンボル）

ネットワーク出力部を作成するときに使用する。
送り先のホスト名をセットする。

・*file-list*　（リスト）

その処理系が処理した仕様データファイル名のリスト。
指定されたファイル名がこの変数になかった場合ファイルをオープンし読み込んで、変数・定数定義、分配部制約、機能部制約および同期部制約をそれぞれのグローバル変数にセットする。ファイルの最後まで読んだらそのファイル名をこの変数に追加する。

・*read-file-list*　（リスト）

その処理系が処理する仕様データファイル名のリスト。
Load-spec または Append-load-spec コマンドで指定されたファイルリストをセットし、変数・定数インスタンス作成モジュールを実行する。
そのモジュールでは、この変数から１つずつ要素を取り出し、処理する。
仕様データファイルの中に、追加ＣＳＬ定義式（(include-constraint ファイル名)）が記述されていた場合、記述されていたファイル名をこの変数に追加する。

・*displayer-name*　（シンボル）

表示系のホスト名。

・*service-status*　（シンボル）

その処理系の状態をセットする。

・*execution-mode*　（シンボル）

実行モード。normal（デフォルト）または debug。

・*execution-mode-list* 〔constant〕　（リスト）

とりうる実行モードのリスト。

《定義》
(defconstant *execution-mode-list* '(normal debug))

・*change-part*　（シンボル）

現在実行している部。
分配部の場合は待状態か評価を開始したかを区別する。
（この変数に類似したもので *part* というグローバル変数もある。〔process〕global-variables.lisp〕ただしこれは、上記の区別をしていない。）

・*part-list* 〔constant〕　　（リスト）

実行部のリスト。

《定義》
(defconstant *part-list* '(decomposing functional superposing))

・*suspend-flag*　　（T または NIL）

サスペンド状態にあるかのフラグ。
初期状態は NIL。

・*step-exec-in*　　（T または NIL）

入力部において入力バッファーからイベントを取り出すかどうかのフラグ。
初期状態は T。

・*step-exec-pro*　　（T または NIL）

仕様処理部において入力キューからイベントを取り出すかどうかのフラグ。
初期状態は T。

8. モジュール構成

ネットワーク仕様処理系全体のモジュール構成を 図8.1 モジュール構成 に示す。
次頁以降で各モジュール毎、デバッグ環境の実現およびネットワークコマンドのやりとりに関連する部分について説明する。

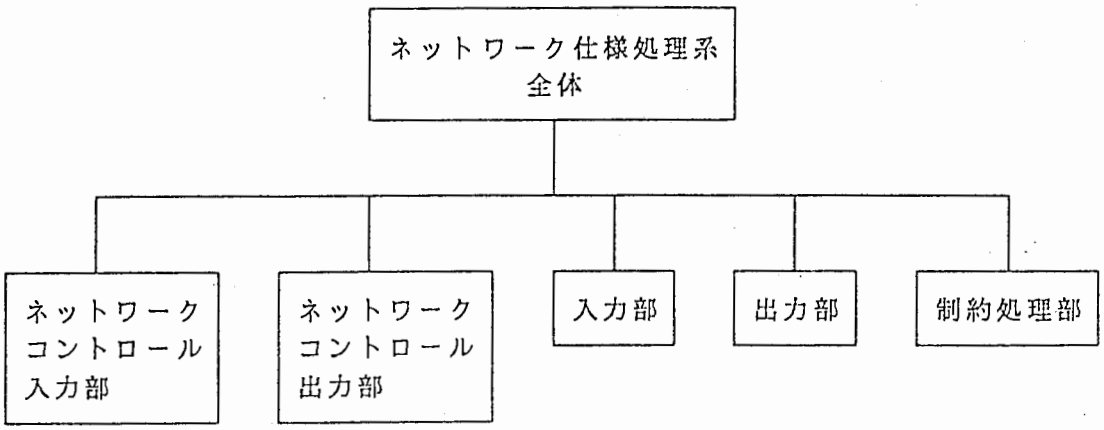


図8.1 モジュール構成

8．1　ネットワークコントロール入力部

以下の処理を行う。
① ネットワークのサーバーを定義し、そのストリームをスロット値として持つ CSL-SERVER フレーバのインスタンスを作成する。
② ストリームから読み込むループにはいる。
（END コマンドが来るまで無限ループになっている。）
③ 読み込みに成功した場合、SUCCESS を、失敗した場合 ERROR を返す。
④ 読み込んだコマンドを解析し、適当な内部処理を行う。

コマンドは、すべてストリングに変換してネットワークを経由してやり取りをする。そのため、コマンドを受け取った場合必要に応じてストリングから元の形に変換しなければならない。（change-command-to-symbol() [network]utility.lisp] で変換する。）

読み込んだコマンドが EVENT なら、ストリングから変換し、入力バッファー（*input-buffer*）の最後に追加する。

それ以外のコマンドならば、init-process() [network]network-io.lisp] で処理する。
init-process() では、各コマンドの内部処理をする関数を実行する。（この関数は、コマンド名と同じ名前になっている。）

以下に、処理系が受け取るコマンドに付いて説明する。（init-process() は、表示系と処理系で共通である。ここでは、処理系が受け取るコマンドについてのみ説明する。）

コマンド名（関数名）　　[コマンド情報]
　説明

①Append-Load-Spec()　　[id　追加する仕様データファイルリスト]

　idおよびサービス状態チェック後、現在読み込んでいる仕様データファイルの定数・変数定義および各制約に、指定されたファイルから読み込んだそれを追加して、変数・定数のインスタンスを新たにつくり直す。
　記述エラーがなければ main-process-start() を実行する。
　記述エラーがあれば、read-spec-error() を実行する。

　main-process-start()

　　デバッグ用フラグを初期化し、サービス状態 *service-status* を :enable にする。また、入力プロセス、出力プロセスおよび制約処理プロセスを起動し、service-enable コマンドを出力バッファーに入れる。

　read-spec-error()

　　read-error コマンド（送出情報は id、処理した仕様データリストとエラー情報のリスト）を出力バッファーに入れる。

②Init-System()　[id]

idチェック後、現在読み込んでいる仕様データファイルの定数・変数定義および各制約から、変数・定数のインスタンスを新たにつくり直す。
記述エラーがなければ main-process-start() を実行する。
記述エラーがあれば、read-spec-error() を実行する。

③Invoke-Next-Event-Transaction()　[id]

idおよびサービス状態のチェックとサスペンド状態にあるかをチェックした後、以下の処理を行う。

1)　制約処理部が分配部の入力待状態にある場合（*change-part* が :idling の場合）
　1)-1　入力キューに処理待イベントがあれば、*step-exec-pro* を T にする。
　　（これによって制約処理部では入力キューからイベントを１つ取りだして分配制約の評価を開始する。）
　1)-2　入力キューが空であれば、*step-exec-in* と *step-exec-pro* を T にする。
　　（これにより、入力部は入力バッファーからイベントを取り出して当該処理系に対する新たなイベントであれは入力キューに入れる。制約処理部では入力キューからイベントを１つ取りだして分配制約の評価を開始する。）

2)　制約処理部が分配部の入力待状態以外である場合
　step-exec-in を T にする。
　　（これにより、入力部は自プロセスへの返答のイベントが見つかるか（見つかったら値をセットする）、または入力バッファーが空になるまでバッファーからイベントを取り出す。この場合当該処理系に対する新たなイベントがあれは入力キューに入れる。）

④Load-Spec()　[id 仕様データファイルリスト]

サービス状態チェック後、*self-process-id* に指定されたidをセットする。
その後、指定された仕様データファイルを読み込んで変数・定数のインスタンスを新たにつくる。
記述エラーがなければ main-process-start() を実行する。
記述エラーがあれば、read-spec-error() を実行する。

⑤Resume-Event-Transaction()　[id]

idおよびサービス状態のチェックとサスペンド状態にあるかをチェックした後サスペンド状態を解除する。すなわち*suspend-flag* に NIL を、
step-exec-in および *step-exec-pro* に T をセットする。

⑥Server-Host　[表示系のホスト名]

表示系のホスト名を *displayer-name* にセットする。またネットワーク管理フレーバのインスタンスを作成しテーブル *csl-system-consiguration-table* にセットする。
ネットワークコントロール出力プロセスを起動し、client-ready コマンドを出力バッファーに入れ、サービス状態 *service-status* を :ready にする。

⑦Set-Mode()　[id　セットするモード]

idおよびサービス状態をチェックした後、モードを指定されたものに変更する。
指定されたモードがデバッグモードであれば current-exec-part コマンドを出力キューに入れる。

⑧Set-Variables-Value()　[id セットする変数、ＶＩＥＷおよび値のリスト]

idおよびサービス状態をチェックした後、指定された変数について値を変更する。そしてその結果を result-set-variables-value コマンドで表示系に返す。第２パラメータとして以下のセットした結果のリストを渡す。

１つの変数とＶＩＥＷの組について、スコープをチェックし、
i)　指定されたＶＩＥＷが有効であれば変更した変数とＶＩＥＷおよび値（リスト）のリスト
ii) エラーがあれば、変数と指定したＶＩＥＷおよびエラーの理由（ストリング）のリスト

⑨Show-Constraints()　[id 実行部]

idおよびサービス状態をチェックした後、指定された実行部の制約式を constraints コマンドで表示系に返す。
分配制約・機能制約・同期制約はそれぞれ、*decomposing-part* *functional-part* *superposing-part* にセットされている。

⑩Show-Exec-part()　[id]

idおよびサービス状態をチェックした後、現在実行している部を current-exec-part コマンドで表示系に返す。

⑪Show-Variables-Value()　[id]

idおよびサービス状態をチェックした後、当該処理系の setvar で定義された変数の変数名とＶＩＥＷおよび値のリストを第２パラメータとした variables-value コマンドで表示系に返す。

⑫Suspend-Event-Transaction()　[id]

idおよびサービス状態をチェックした後、サスペンド状態にする。すなわち、*suspend-flag* に T を、*step-exec-in* および *step-exec-pro* に NIL をセットする。

8．2　ネットワークコントロール出力部

ネットワークのクライアントを定義し、そのストリームを、ネットワーク管理テーブル *csl-system-consiguration-table* の表示系ホストのストリーム情報としてセットする。

ネットワークコントロール出力部は１つのプロセスであり、以下の処理を行う無限ループである。

すなわち、出力バッファー *output-buffer* に送出情報が入っていたら、１つ取りだして、情報の先頭のストリームに情報のCDR部をストリングに変換したものを送り、受け取り成功か失敗かの返答を待つ。成功するまで送る。

8．3　入力部

入力部は１つのプロセスであり、以下処理を行う無限ループである。

すなわち、*step-exec-in* が T の場合、入力バッファー *input-buffer* からイベント読み込み処理 input-process-internal() 〔communication〉in-process.lisp〕 を行う。その後、*suspend-flag* が T の場合、*step-exec-in* を NIL にする。

このフラグ操作によって、サスペンド、ステップ実行を実現する。

イベント読み込み処理では、以下の処理を行う。

入力バッファーにイベントが入っていれば、１つ取りだして、メッセージidにより当該処理系に対する返答か新たなイベントか、あるいはそれ以外かを以下のやり方で調べる。

① プロセスidが自プロセス以外ならば、新たなイベント。

② プロセスidが自プロセスであり、かつプロセスidとラウンドidの組が実行リスト *execution-list* にあれば自プロセスへの返答。

③ 上記以外は捨ててよいイベント

8．4　出力部

出力キュー *output-que* にコマンド（イベントを含む）が入っていた場合、取りだして以下の処理を行う。

① イベントの場合

①-1 自プロセスへのイベント

出力情報の先頭に表示系のストリームを追加して出力バッファーに追加する。

①-2 外部プロセスへのイベント

1)出力情報の先頭に表示系のストリームを追加して出力バッファーに追加する。

2)実行リスト *execution-list* にメッセージidを追加する。

3)タイマーが設定されていれば、タイマーを起動する。

② イベント以外のコマンドの場合

出力情報の先頭に表示系のストリームを追加して出力バッファーに追加する。

8．5　制約処理部

制約処理部は１つのプロセスであり、以下の処理を行う無限ループである。

すなわち、分配部の入力待状態において、*step-exec-pro* が T の場合、入力キュー *input-que* からイベントを１つ取りだして値をセットし分配制約の評価を開始する。分配部、機能部、同期部の処理を終えて再度制御が分配部入力待状態に戻ったときに、*suspend-flag* が T の場合、*step-exec-in* を NIL にする。

このフラグ操作によって、サスペンド、ステップ実行を実現する。

制御が分配部の入力待状態に移った時点、分配制約の評価開始に移った時点、機能部に移った時点および同期部に移った時点で、デバッグモードであれば（*execution-mode* が debug である場合）current-exec-part コマンドを出力キューに追加する。

制約評価処理を行っている際、制約式の条件部をチェックし、真になる制約式が見つかった時点で、デバッグモードであればその制約式を fired-rule コマンドの引数として出力キューに追加する。

付録2

ネットワーク仕様処理系

ソースリスト

1990年 8月

目次

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: (CSL) -*-


;;;
;;; FLAVOR DEFINITION
;;;


;;; This flavor conrol all variables and constanit.

(defflavor variables
        ((event-variables     nil)
         (adress-variables    nil)
         (status-variables    nil)
         (parameter-variables nil)
         (constant            nil))
        ()
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)

;;;

(defflavor view-variable
        ((view    nil)
         (value   :unbound))
        ()
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)


;;; Event variable's flavor definition.

(defflavor event-variable
        ((part  nil)
         (from  :unbound)
         (value :unbound))
        ()
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)

;;; Adress variable's flavor definition.

(defflavor adress-variable
        ()
        (view-variable)
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)

;;; Status variable's flavor definition.

(defflavor status-variable
        ()
        (view-variable)
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)

;;; Parameter variable's flavor definition.

(defflavor parameter-variable
        ((value   :unbound))
        ()
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)

;;; Constant flavor definition.

(defflavor constant
        ()
        (view-variable)
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Following variables to use GLOBAL VARIABLES
;;;

;;; Set top flavor instance.

(defvar *variables* nil)                    ; Set flavor VARIABLES instance

(defvar *event-variables* nil)              ; Set flavor EVENT-VARIABLES instance

;;; Execution list.

(defvar *execution-list* nil)

;;; Process id

(defvar *self-process-id* nil)

(defvar *return-process-id* nil)

(defvar *return-process* nil)

(defvar *device-id* 'device)

;;; Round id

(defvar *round-id* 0)

(defvar *return-round-id* nil)

;;; Stack for Constant.

(defvar *constant-variable* nil)

;;; Reserved word.

(defconstant *reserved-words* '(:@self :@dcmp :@func :@supr :@timer :@dev :@intr :timing rele
asetiming))

(defconstant *self-process-variables* '(:@self :@dcmp :@func :@supr :@timer :@dev :@intr))

(defconstant *interactive-part* '(:interactive :decomposing :superposing))

(defconstant *@intr* '(:@dcmp :@supr))

(defconstant *self-process-part* '(:@dcmp :@func :@supr))

;;; Stack for Body in :decomposing and :functional.

(defvar *sentence* nil "stack action part")

;;; Stack for Functional-event and Superposing-event

(defvar *event-to-functional* nil "stack event to functional")

(defvar *event-to-superposing* nil "stack event to superposing")

;;; Some Flags used in program.

(defparameter *part* nil)

(defvar *error* nil)

(defvar *csl-flow* nil)

(defvar *constraint* nil)
```

```
;;; Event variables and parameter variables stacking at interpreting rule
(defvar *p-variables* nil  "PARAMETER variables")
(defvar *e-variables* nil  "EVENT variables")
(defvar *unify-p-variable* nil)
(defvar *unify-var-value*  nil)
(defvar *variable-pair-list* nil)              ; for Superposing-part  (:@intr)

;;;
(defvar *decomposing-part* nil)
(defvar *functional-part*  nil)
(defvar *superposing-part* nil)

;;; Read error buffer
(defvar *read-error-buffer* nil)

;;; For test
(defvar *standard-output-flag* nil)
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Make all variables' instance.
;;;

(defun make-variables-instance (f-list &optional flg)
  (setq *variables* (make-instance 'variables))
  (reset-make-instance-buffer-flag)
  (or (read-csl f-list flg) *error* (make-variables-instance-internal)))


;;; Parsing variable or constant definition

(defun make-variables-instance-internal ()
  (reset-variables-slots *variables*)
  (make-constant-variable)
  (make-decomposing-part)
  (make-functional-part)
  (make-superposing-part)
  (setq *constraint* nil))

(defun make-constant-variable ()
  (dolist (var *constant-variable*)
    (setq *constraint* var)
    (parse-constant-variable var)))

(defun parse-constant-variable (var)
  (let ((view (second var))
        (vars (cddr var)))
    (case (car var)
      (setvar   (parse-variable-internal view vars))
      (setconst (parse-constant-internal view vars)))))

(defun parse-variable-internal (view vars)
  (do* ((var (pop vars) (pop vars))
        (val (pop vars) (pop vars)))
       ((and (null var) (null val)))
    (case (variable-type var)
      (:status (set-status-variable var val view))
      (:adress
        (if (eql var :@dev)
            (setq *device-id* val)
            (set-adress-variable var val view)))
      (t (error-call 1 var)))))

(defun parse-constant-internal (view vars)
  (do* ((var (pop vars) (pop vars))
        (val (pop vars) (pop vars)))
       ((and (null var) (null val)))
    (case (variable-type var)
      ((:adconst :constant) (set-constant var val view))
      (t (error-call 2 var)))))


;;; Parsing rule

(defun make-decomposing-part ()
  (dolist (rule *decomposing-part*)
    (setq *constraint* rule)
    (parse-rule rule :decomposing)))

(defun make-functional-part ()
  (dolist (rule *functional-part*)
    (setq *constraint* rule)
    (parse-rule rule :functional)))

(defun make-superposing-part ()
  (dolist (rule *superposing-part*)
    (setq *constraint* rule)
    (parse-rule rule :superposing)))

(defun parse-rule (rule part)
  (let ((condition (car rule))
        (action    (third rule)))
    (parse-condition condition part)
    (parse-action action part)))
```

```
(defun parse-condition (condition part)
  (let ((count 1))
    (dolist (cond condition)
      (cond ((and (eql cond 'or) (= count 1))
             (parse-condition (cdr condition) part))
            ((or (eql (car cond) 'and) (eql (car cond) 'or))
             (parse-condition (cdr cond) part))
            ((listp (car cond))
             (parse-condition cond part))
            ((and (symbolp cond) (≠ count 1))
             (error-call 5 condition) (return nil))
            (t (parse-condition-internal cond part)))
      (setq count (1+ count)))))

(defun parse-condition-internal (condition part)
  (let* ((cond (if (eql (car condition) '¬) (cdr condition) condition))
         (var  (pop cond)))
    (case (variable-type var)
      (:event     (set-event-variable var part))
      ((:status :adress :constant) (or (scope-check var part) (error-call 6 var)))
      (:parameter
        (if (eql (variable-type (second cond)) :adress)
            (error-call 5 condition)
            (set-parameter-variable var))))
    (dolist (term cond)
      (case (variable-type term)
        (:parameter (set-parameter-variable term))
        ((:status :adress :constant) (or (scope-check term part) (error-call 6 term)))))))

(defun parse-action (actions part)
  (dolist (action actions)
    (if (listp action)
        (parse-action-internal action part)
        (error-call 11 actions))))

(defun parse-action-internal (action part)
  (dolist (term action)
    (if (member term '(and or =>))
        (error-call 10 action)
        (case (variable-type term)
          ((:status :adress :constant)
           (if (boundp term)
               (or (scope-check term part) (error-call 6 term))
               (error-call 9 term)))
          (:parameter (set-parameter-variable term))))))  ;;; ?


;;;

(defun set-event-variable (var part)
  (if (event-variable-existp var)
      (push-event-part (symbol-value var) part)
      (set var (make-instance 'event-variable :part `(,part)))
      (push-event-variable var)))

(defun set-adress-variable (var val view)
  (set-view-variable var val view 'adress-variable :adress))

(defun set-status-variable (var val view)
  (set-view-variable var val view 'status-variable :status))

(defun set-constant (var val view)
  (set-view-variable var val view 'constant :constant))

(defun set-view-variable (var val view flavor-name key)
  (if (variable-exist-check *variables* val)
      (set-view-variable-1 var val view flavor-name key)
      (set-view-variable-2 var val view flavor-name key)))

(defun set-view-variable-1 (var val view flavor-name key)
  (let ((value (car (expand-variables `(,val)))))
    (set-view-variable-2 var value view flavor-name key)))

(defun set-view-variable-2 (var val view flavor-name key)
  (if (variable-exist-check *variables* var key)
      (set-view-variable-internal var val view)
;       (case (view-check var view)
;         (2 (push-view-value (symbol-value var) val view))
```

```
;       (3 (change-view-value (symbol-value var) val view)))
      (set var (make-instance flavor-name :view '(,view) :value '((,val))))
      (push-variable *variables* var key)))

(defun set-view-variable-internal (var val view)
  (let ((view-list nil)
        (value-list nil))
    (loop for setted-view  in (view-variable-view (symbol-value var))
          for setted-value in (view-variable-value (symbol-value var))
          for flag = (view-check-2 setted-view view)
          when (eql flag 1)
            do (return-from set-view-variable-internal)
          when (eql flag 2)
            do
              (push setted-view  view-list)
              (push setted-value value-list))
    (push view view-list)
    (push '(,val)  value-list)
    (change-view-value-2 (symbol-value var) view-list value-list)))

(defun set-parameter-variable (var)
  (unless (parameter-variable-existp var)
    (set var (make-instance 'parameter-variable))
    (push-parameter-variable var)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-


;;;
;;; Control flow
;;;


;;; top function

(defun process-start (&optional flg)
  (reset-buffers-and-id-in-process-1 flg)
  (setq *change-part* :idling)
  (if (debug-mode-p) (send-current-exec-part *change-part*))
  (loop for que-flag = nil
        do
    (and *step-exec-pro*
         (setq que-flag (process-start-internal)))
    (and *suspend-flag*
         que-flag
         (setq *step-exec-pro* nil))))

(defun process-start-internal ()
  (let (input-event)
    (when *input-que*
      (reset-buffers-and-id-in-process-2)
      (without-interrupts
        (setq input-event (pop *input-que*)))
      (and (apply #'set-input-event input-event)
           (processing-1)
           (processing-2))
      (setq *change-part* :idling)
      (if (debug-mode-p) (send-current-exec-part *change-part*))
      T)
    ))

;(defun process-start ()
;  (reset-buffers-and-id-in-process)
;  (setq *self-process-id* 'SELF        *round-id* 1) ;;; for test
;  (and (processing-1) (processing-2)))

(defun processing-1 ()
  (decomposing-part))

(defun processing-2 ()
  (setq *csl-flow* nil)
  (set-functional-event)
  (functional-part)
  (setq *event-to-functional* nil)
  (set-superposing-event)
  (superposing-part)
  (setq *event-to-superposing* nil)
  (if (eql *csl-flow* :functional)
      (processing-2)
      (setq *part* :decomposing
            *change-part* :decomposing)))


;;; Control the decomposing part

(defun decomposing-part ()
  (setq *part* :decomposing
        *sentence* nil
        *change-part* :decomposing)
  (if (debug-mode-p) (send-current-exec-part *change-part*))
  (interpret-decomposing))

(defun interpret-decomposing ()
  (let ((fired (interpret-decomposing-internal)))
    (reset-all-decomposing-event-variables)
    (execute-stacked-rhs *sentence*)
;    (reset-all-decomposing-event-variables)
    fired))

(defun interpret-decomposing-internal ()
  (let (fired-flag)
    (dolist (rule *decomposing-part* fired-flag)
      (and (interpret-rule rule :decomposing) (setq fired-flag t)))))
```

```
;;; Control the functional part

(defun functional-part ()                          ;
  (setq *part* :functional
        *sentence* nil
        *change-part* :functional)
  (if (debug-mode-p) (send-current-exec-part *change-part*))
  (interpret-functional))

(defun interpret-functional ()
  (interpret-functional-internal)
  (reset-all-functional-event-variables)
  (execute-stacked-rhs *sentence*)
;  (reset-all-functional-event-variables)
  )

(defun interpret-functional-internal ()
  (dolist (rule *functional-part*)
    (interpret-rule rule :functional)))


;;; Control the superposing part

(defun superposing-part ()
  (setq *part* :superposing
        *change-part* :superposing)
  (if (debug-mode-p) (send-current-exec-part *change-part*))
  (interpret-superposing))

(defun interpret-superposing ()
  (loop for i from 1
        do
    (if (event-variables-boundp *part*)
        (interpret-superposing-internal)
        (return nil)))
;(reset-all-event-variables-set-flag)
  (reset-all-superposing-event-variables)
  (reset-execution-list))

(defun interpret-superposing-internal ()
  (dolist (rule *superposing-part*)
;    (interpret-rule rule :superposing)))
    (interpret-supr-rule rule)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Rule interpreter
;;;


;;; Interpret Rule

(defun interpret-rule (rule part)
  (setq *p-variables* nil
        *e-variables* nil)
  (let (fired
        (condition (car rule))
        (action    (third rule)))
    (reset-parameter-variables)
    (when (interpret-lhs condition)
      (if (debug-mode-p) (send-fired-rule rule))
      (setq fired t)
      (and *standard-output-flag* (format t "~%!!! Fired rule ~A~%" rule)) ;;; test
      (if (eql part :superposing) (interpret-rhs action) (expand-rhs-part action))
      (and (eql part :superposing) (reset-stacked-event-variables *e-variables* *part*))
      fired)))


;;; Interpret LHS

(defun interpret-lhs (conditions)
  (loop for condition in conditions
        unless (interpret-condition condition)
          do (return nil)
        finally (return t)))

(defun interpret-condition (condition)
  (if (listp (car condition))
      (interpret-and-condition condition)
      (case (car condition)
        (or  (interpret-or-condition (cdr condition)))
        (and (interpret-and-condition (cdr condition)))
        (t   (interpret-condition-internal condition)))))

(defun interpret-and-condition (conditions)
  (interpret-lhs conditions))

(defun interpret-or-condition (conditions)
  (let (satisfied)
    (loop for condition in conditions
          when (interpret-condition condition)
            do (setq satisfied t)
          finally (return satisfied))))

(defun interpret-condition-internal (condition)
  (let* ((not-flag  (eql (car condition) '¬))
         (condition (if not-flag (cdr condition) condition))
         (variable  (car condition))
         (values    (cdr condition)))
    (case (variable-type variable)
      (:event
;       (and (eql variable 'timeout) (pop values))
       (and (interpret-event-condition not-flag variable (car values) (cdr values))
            (push condition *e-variables*)))
      (:status    (interpret-status-condition not-flag variable values))
      (:adress    (interpret-adress-condition not-flag variable values))
      (:parameter
        (interpret-parameter-condition not-flag variable values)))))

(defun interpret-event-condition (not-flag variable-name sent-part values)
  (let ((check (check-event-condition variable-name sent-part values)))
    (if not-flag (not check) check)))

(defun check-event-condition (variable-name sent-part values)
  (case *part*
    ((:decomposing :functional)
     (check-event-condition-df (symbol-value variable-name) sent-part values))
    (:superposing
      (check-event-condition-supr (symbol-value variable-name) sent-part values))))

(defmacro condition-values-check (variable-value parameter-list)
```

```
  `(parameter-matching ,variable-value ,parameter-list))

(defmethod (check-event-condition-df EVENT-VARIABLE)(sent-part values)
  (let ((val       (if (eql value :unbound) NIL value))
        (sent-part (cond ((member sent-part *reserved-words*) sent-part)
                         ((eql (variable-type sent-part) :parameter) sent-part)
                         (t (car (expand-variables `(,sent-part)))))))
    (and (not (eql from :unbound))
         (cond ((eql from sent-part) t)
               ((eql sent-part :@intr) (member from *@intr*))
               ((eql (variable-type sent-part) :parameter)
                (set-parameter-value sent-part `(,from)) T)
               (t NIL))
         (if values
             (condition-values-check val values)
             T))))

(defmethod (check-event-condition-supr EVENT-VARIABLE)(sent-part values)
  (let* ((sent-part (cond ((member sent-part *reserved-words*) sent-part)
                          ((eql (variable-type sent-part) :parameter) sent-part)
                          (t (car (expand-variables `(,sent-part))))))
         (pos       (cond ((eql from :unbound) nil)
                          ((eql (variable-type sent-part) :parameter) 0)
                          ((not (eql from :unbound))
                           (if (eql sent-part :@timer)  ;ogata
                               (get-timer-pos self (car (expand-variables values)))     ;ogat
a
                               (position sent-part from :test #'pos-test)))))
         val)
    (when (and (not (eql from :unbound)) pos)
      (and (eql (variable-type sent-part) :parameter)
           (set-parameter-value sent-part `(,(car from))))
      (if (eql sent-part :@timer)               ;ogata
          T                                     ;ogata
          (setq val (nth pos value))
          (if values
              (condition-values-check val values)
              T)))))


(defun interpret-status-condition (not-flag variable-name values)
  (let ((check (interpret-status-condition-internal variable-name values)))
    (if not-flag (not check) check)))

(defun interpret-status-condition-internal (variable-name values)
  (let ((val (get-status-value variable-name *part*)))
    (unless (eql val :unbound) (parameter-matching val values))))
;    (unless (eql val :unbound) (unify val values))))

(defun interpret-adress-condition (not-flag variable-name values)
  (let ((check (interpret-adress-condition-internal variable-name values)))
    (if not-flag (not check) check)))

(defun interpret-adress-condition-internal (variable-name values)
  (let ((val (get-adress-value variable-name *part*)))
    (unless (eql val :unbound) (parameter-matching val values))))
;    (unless (eql val :unbound) (unify val values))))

(defun interpret-parameter-condition (not-flag variable-name values)
  (let ((check (interpret-parameter-condition-internal variable-name values)))
    (if not-flag (not check) check)))

(defun interpret-parameter-condition-internal (variable-name values)
  (and (parameter-variable-boundp (symbol-value variable-name))
       (parameter-matching (get-parameter-value variable-name) values)))
;       (equal (get-parameter-value variable-name) values)))


;(defmacro condition-values-check (variable-value parameter-list)
;  `(parameter-matching ,variable-value ,parameter-list))
;  `(unify ,variable-value ,parameter-list))


;;; Expand RHS

(defun expand-rhs-part (actions)
  (dolist (action actions)
    (setq *sentence* `(,@*sentence* ,(expand-rhs-part-internal action)))))
```

```
(defun expand-rhs-part-internal (action-body)
  (multiple-value-bind (action timer)
      (separate-action action-body)
    (setq timer (if timer timer 0))
    (let ((expanded-action (expand-variables (cdr action)))
          (type (variable-type (car action))))
      (if (eql type :event)
          `(,(car action) ,@expanded-action :timing ,timer)
          `(,(car action) ,@expanded-action)))))


;;; Interpret RHS

(defun interpret-rhs (actions)
  (let((actions (loop for action in actions
                      collect (expand-rhs-part-internal action))))
    (dolist (action actions)
      (interpret-action action))))

(defun interpret-action (action)
  (if (eql (car action) 'releasetiming)
      (execute-releasetiming-action action)
      (execute-rhs action)))


;;; called from the Flow-Control module.

(defun execute-stacked-rhs (bodys)
  (dolist (body bodys)
    (execute-rhs body)))

;;;


(defun execute-rhs (body)
  (let ((var (car body)))
    (case (variable-type var)
      (:event     (execute-event-body body *part*))
      (:status    (execute-status-body body *part*))
      (:adress    (execute-adress-body body *part*))
      (:constant  (execute-constant-body body *part*))
      (:parameter (execute-parameter-body body)))))

(defun execute-event-body (body part)
  (multiple-value-bind (action-body timer)
      (separate-action body)
    (let* ((var (pop action-body))
           (to  (pop action-body))
           (from (change-part part)))
      (if (send-self-process-p to)
          (set-event-value var to from action-body)
          (set-and-send-event-variable var to from action-body timer)))))

(defun execute-status-body (body part)
  (set-status-value (car body) (cdr body) part))

(defun execute-adress-body (body part)
  (set-adress-value (car body) (cdr body) part))

(defun execute-constant-body (body part)
  (set-constant-value (car body) (cdr body) part))

(defun execute-parameter-body (body)
  (set-parameter-value (car body) (cdr body)))

(defun execute-releasetiming-action (action)
  (let ((to-id (car (expand-variables (cddr action)))))
    (remake-execution-list to-id)
    (reset-timer to-id)))


;;; Interpret Rule for superposing part

(defun interpret-supr-rule (rule)
  (setq *e-variables* nil
        *variable-pair-list* nil)
  (let* (fired
         (condition (car rule))
```

```
        (action    (third rule))
        (variable-pair-list (get-variable-pair-list condition)))
    (if variable-pair-list
        (setq fired (interpret-supr-rule-1 variable-pair-list rule condition action))
        (setq fired (interpret-supr-rule-internal rule condition action))
        fired)))

(defun interpret-supr-rule-1 (variable-pair-list rule condition action)
  (let* (fired
         (fired (dolist (variable-pair variable-pair-list fired)
                  (preset-event-variable-value variable-pair)
                  (if (interpret-supr-rule-internal rule condition action)
                      (setq fired T)
                      (push variable-pair *variable-pair-list*)))))
    (when *variable-pair-list*
      (episet-event-variable-value *variable-pair-list*))
    fired))

(defun interpret-supr-rule-internal (rule condition action)
  (reset-parameter-variables)
  (when (interpret-lhs condition)
    (if (debug-mode-p) (send-fired-rule rule))
    (reset-stacked-event-variables *e-variables* *part*)
    (interpret-rhs action)
    T))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Unify variable values to rule expression parameter.
;;;


(defun unify (variable-value parameter-list)
  (setq *unify-p-variable* nil
        *unify-var-value*  nil)
  (let ((variable-value (expand-parameter-list variable-value))
        (parameter-list (expand-parameter-list parameter-list)))
    (cond ((and (not variable-value) (not parameter-list)) t)
          ((and (not variable-value) parameter-list) nil)
          ((and variable-value (not parameter-list)) nil)
          ((and (= (length variable-value) 1)
                (eql (variable-type (car variable-value)) :parameter)
                (= (length parameter-list) 1)
                (eql (variable-type (car parameter-list)) :parameter))
           NIL)
          ((and (= (length variable-value) 1)
                (eql (variable-type (car variable-value)) :parameter))
           (unify-parameter-1 variable-value parameter-list))
          ((and (= (length parameter-list) 1)
                (eql (variable-type (car parameter-list)) :parameter))
           (unify-parameter-1 parameter-list variable-value))
          (t (unify-parameter-list variable-value parameter-list)))))

(defun unify-parameter-1 (var value)
  (set-parameter-value (car var) value)
  T)

(defun unify-parameter-list (variable-value parameter-list)
  (let (para-flag)
    (unless (eql :unbound variable-value)
      (dolist (para parameter-list)
        (if (and (eql (variable-type para) :parameter)
                 (eql (get-parameter-value para) :unbound))
            (if para-flag
                (return nil)
                (setq para-flag T)
                (push para *unify-p-variable*))
            (when (eql (setq variable-value (unify-symbol para variable-value)) :unmatch)
              (return nil))
            (setq para-flag nil)))
      (cond ((and variable-value (= (length *unify-p-variable*) 1))
             (unify variable-value *unify-p-variable*))
            (variable-value NIL)
            (T (stack-check))))))

(defun unify-symbol (para value-list)
  (do ((val-list value-list (cdr val-list))
       (stack nil))
      ((or (not val-list) (equal para (car val-list)))
       (if (not val-list)
           :unmatch
           (and stack (push (reverse stack) *unify-var-value*))
           (cdr val-list)))
    (push (car val-list) *unify-var-value*)))

(defun stack-check ()
  (let ((p-vars     (reverse *unify-p-variable*))
        (var-values (reverse *unify-var-value*)))
    (when (eql (length p-vars) (length var-values))
      (loop for p-var in p-vars
            for var-value in var-values
            do
        (set-parameter-variable p-var)
        (if (eql (get-parameter-value p-var) :unbound)
            (set-parameter-value p-var `(,var-value))  ;;; really ok?
            (or (equal (get-parameter-value p-var) var-value) (return nil))))
      T)))

;;;

(defun parameter-matching (variable-value parameter-list)
  (cond ((and (= (length variable-value) 1) (= (length parameter-list) 1)
              (not (car variable-value)) (not (car parameter-list))) T)
```

```
        ((and (not variable-value) parameter-list) NIL)
        ((and variable-value (not parameter-list)) NIl)
        (t (parameter-matching-internal variable-value parameter-list))))

(defun parameter-matching-internal (variable-value parameter-list)
  (let (flag)
    (do ((var-list variable-value variable-value)
         (par-list parameter-list parameter-list)
         (variable  (expand-one-parameter (pop variable-value))
                    (expand-one-parameter (pop variable-value)))
         (parameter (expand-one-parameter (pop parameter-list))
                    (expand-one-parameter (pop parameter-list))))
        ((not (and var-list par-list))
         (cond ((and (not variable) parameter) NIL)
               ((not parameter) flag)))
      (cond ((and (eql (variable-type variable)  :parameter)
                  (eql (variable-type parameter) :parameter))
             (return NIL))
            ((eql (variable-type variable) :parameter)
             (setq flag (unify-parameter-2 variable parameter)))
            ((eql (variable-type parameter) :parameter)
             (setq flag (unify-parameter-2 parameter variable)))
            (T (unless (setq flag (equal variable parameter)) (return NIL)))))))

(defun expand-one-parameter (variable)
  (first (expand-parameter-list '(,variable))))

(defun unify-parameter-2 (var value)
  (set-parameter-value var '(,value))
  T)
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Following functions to set test environment.
;;;


(defun read-csl (f-list flg)
  (unless flg (reset-read-csl-buffer))
  (setq *read-file-list* f-list)
  (loop
    (if *read-file-list*
        (read-csl-internal (pop *read-file-list*))
        (return))))

(defun read-csl-internal (file-name)
  (let ((file-name (make-constraint-file-name file-name)))
    (cond ((not (probe-file file-name))
           (error-call 12 file-name))
          ((not (member file-name *file-list* :test #'equal))
           (with-open-file (infile file-name :direction :input)
             (do ((rules (read infile nil :EOF) (read infile nil :EOF)))
                 ((eql rules :EOF) (push file-name *file-list*))
               (setq rules (to-csl-package rules))
               (case (car rules)
                 (defconstraint      (read-constraint (cdr rules)))
                 ((setvar setconst)  (tail-push *constant-variable* rules))
                 (include-constraint (tail-push *read-file-list* (second rules)))
                 (t                  (error-call 7 rules)))))))))

(defun make-constraint-file-name (file-name)
  (let ((name (if (symbolp file-name) (string file-name) file-name)))
    (cond ((search ":>" name :test #'equal) name)
          ((search ";"  name :test #'equal) name)
          (T (format nil "csl:constraint;~A" name)))))
;    (if (probe-file name)
;       name
;       (setq name (format nil "csl:constraint;~A" name))
;       (if (probe-file name) name))))

(defun read-constraint (rules)
  (case (pop rules)
    (:decomposing (setq *decomposing-part* '(,@*decomposing-part* ,@rules)))
    (:functional  (setq *functional-part*  '(,@*functional-part*  ,@rules)))
    (:superposing (setq *superposing-part* '(,@*superposing-part* ,@rules)))
    (t            (error-call 7 rules))))


(defun to-csl-package (rules)
  (let (str)
    (loop for x in rules
          when (stringp x)
            collect x into return-list
          when (numberp x)
            collect x into return-list
          when (and x (listp x))
            collect (to-csl-package x) into return-list
          when (and (symbolp x)
                    (setq str (format nil "~s" x))
                    (equal (subseq str 0 1) ":")
                    (> (length str) 1))
            collect (intern (subseq str 1) 'keyword) into return-list
          when (and (symbolp x)
                    (not (equal (subseq str 0 1) ":")))
            collect (intern str 'csl) into return-list
          finally (return return-list))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; ERROR CODE
;;;

(defvar *erase-code* nil)

(defun error-call (no arg)
  (if (> no 12)
      (error-call-1 no arg)
      (setq *error* t)
      (push (get-error-reason no arg) *read-error-buffer*)))

(defun error-call-1 (no arg)
  (let* ((instance (get-instance :host *displayer-name*))
         (stream (get-info instance :stream))
         (error (get-error-reason no arg)))
    (push-output-buffer
      `(,stream execution-error ,error))))

(defun get-error-reason (no arg)
  (setq *erase-code* "")
  (case no
    ((1 2) (format nil "Illegal ~a declared in ~A." arg (erase-cr-code *constraint*)))
    ((3 4) (format nil "~A view illegaly in ~A." arg (erase-cr-code *constraint*)))
    (5     (format nil "Illegal format condition ~Ain ~A." (erase-cr-code arg) (erase-cr-code
 *constraint*)))
    (6     (format nil "Variable ~A scope illegal in ~A." arg (erase-cr-code *constraint*)))
    (7     (format nil "Illegal CSL format ~A." (erase-cr-code arg)))
    (8     (format nil "Illegal view ~A in ~A." arg (erase-cr-code *constraint*)))
    (9     (format nil "Undefined variable ~A is used in ~A." arg (erase-cr-code *constraint*
)))
    (10    (format nil "Illegal format action ~A in ~A." (erase-cr-code arg) (erase-cr-code *
constraint*)))
    (11    (format nil "Illegal format ~A in ~A." (erase-cr-code arg) (erase-cr-code *constra
int*)))
    (12    (format nil "The file ~A was not found." arg))))


;;;

(defun erase-cr-code (arg)
  (cond ((and arg (listp arg))   ;(and (listp arg) (listp (car arg)))
         (setq *erase-code* (format nil "~A(" *erase-code*))
         (dolist (x arg)
           (erase-cr-code x))
         (setq *erase-code* (format nil "~A)" *erase-code*)))
;        ((and (listp arg) (not (listp (car arg))))
;         (setq *erase-code* (format nil "~A(" *erase-code*))
;         (dolist (x arg)
;           (erase-cr-code x))
;         (setq *erase-code* (format nil "~A)" *erase-code*)))
;         (setq *erase-code* (format nil "~A~A" *erase-code* arg)))
        ((stringp arg)
         (setq *erase-code* (format nil "~A ~S " *erase-code* arg)))
        (t (setq *erase-code* (format nil "~A ~A " *erase-code* arg)))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-


;;;
;;; Following functions and methods for utility from all modules.
;;;


;;; Check variable type

(defun variable-type (var)
  (let* ((variable (format nil "~A" var))
         (first-char (subseq variable 0 1))
         (two-char   (if (> (length variable) 1) (subseq variable 0 2) "")))
    (cond ((equal two-char   "%@"):constant)     ;;; :adconst)
          ((equal first-char "@") :adress)
          ((equal first-char "$") :status)
          ((equal first-char "%") :constant)
          ((equal first-char "*") :parameter)
          (t :event))))
```

```
;;; Some predicate

;;;

(defmacro send-self-process-p (to)
  `(member ,to *self-process-variables*))

(defmacro self-process-part-p (to)
  `(member ,to *self-process-part*))

;;; Variable exist predicate

(defmacro event-variable-existp (var)
  `(variable-exist-check *variables* ,var :event))

(defmacro status-variable-existp (var)
  `(variable-exist-check *variables* ,var :status))

(defmacro adress-variable-existp (var)
  `(variable-exist-check *variables* ,var :adress))

(defmacro parameter-variable-existp (var)
  `(variable-exist-check *variables* ,var :parameter))

(defmacro constant-existp (const)
  `(variable-exist-check *variables* ,const :constant))

(defun variablep (var)
  (without-interrupts
    (variable-exist-check var *variables*)))

(defmethod (variable-exist-check VARIABLES)(var &optional key)
  (car (if key
           (case key
             (:event     (member var event-variables))
             (:status    (member var status-variables))
             (:adress    (member var adress-variables))
             (:parameter (member var parameter-variables))
             (:constant  (member var constant)))
           (member var `(,@event-variables ,@status-variables ,@adress-variables ,@parameter-
variables ,@constant)))))

(defmethod (parameter-variable-boundp PARAMETER-VARIABLE)()
  (not (eql value :unbound)))

(defmacro parameter-variable-stacked-p (variable)
  `(or (member ,variable *p-variable*) (push ,variable *p-variable*)))

(defun event-variables-boundp (&optional part)
  (event-variables-boundp-internal *variables* part))

(defmethod (event-variables-boundp-internal VARIABLES)(&optional part)
  (loop for var in event-variables
        when (event-variable-boundp (symbol-value var) part)
          do (return t)
        finally (return nil)))

(defmethod (event-variable-boundp EVENT-VARIABLE)(&optional used)
  (if used
      (and (member used part) (not (eql value :unbound)))
      (not (eql value :unbound))))

(defun pos-test (item sequence)
  (if (eql item :@intr)
      (member sequence '(:@dcmp :@supr))
      (eql item sequence)))
```

```
;;; Check functions

(defun view-check (var view)
  ;;; return value = 3 -> change view and value
  ;;; return value = 2 -> push view and value
  (let* ((setted-view (view-variable-view (symbol-value var)))
         (setted-view-point (get-view-point (car setted-view)))
         (view-point (get-view-point view)))
    (cond ((> setted-view-point view-point)
           (if (and (eql (car setted-view) :functional) (= view-point 3))
               2 1))
          ((= setted-view-point view-point)
           (if (or (scope-subcheck-2 (car setted-view) view)
                   (scope-subcheck-3 (car setted-view) view))
               2 1))
          ((< setted-view-point view-point)
           (if (scope-subcheck-1 (car setted-view) view) 2 3)))))

(defun view-check-2 (setted-view view)
  ;;; return value = 3 -> erase view and value
  ;;; return value = 2 -> push view and value
  (let* ((setted-view-point (get-view-point setted-view))
         (view-point (get-view-point view)))
    (cond ((> setted-view-point view-point)
           (if (and (eql setted-view :functional) (= view-point 3))
               2 1))
          ((= setted-view-point view-point)
           (if (or (scope-subcheck-2 setted-view view)
                   (scope-subcheck-3 setted-view view))
               2 1))
          ((< setted-view-point view-point)
           (if (scope-subcheck-1 setted-view view) 2 3)))))

(defun scope-check (var part)
  (or (send-self-process-p var)
      (and (variable-exist-check *variables* var)
           (let ((setted-view (view-variable-view (symbol-value var))))
             (scope-check-internal setted-view part)))))

(defun scope-check-internal (setted-views part)
  (loop for setted-view in setted-views
        when (let ((setted-view-point (get-view-point setted-view))
                   (view (get-view-point part)))
               (cond ((and (eql setted-view :functional) (= view 3)) nil)
                     ((eql setted-view part) t)
                     ((> setted-view-point view) t)
                     ((and (= setted-view-point view) (scope-subcheck-1 setted-view view)) t)
                     (t nil)))
          do (return t)
        finally (return nil)))

(defun scope-subcheck-1 (setted-view view)
  ;;; one view equal functional part, another inclued interactive part?
  (or (and (member setted-view *interactive-part*) (eql view :functional))
      (and (eql setted-view :functional) (member view *interactive-part*))))

(defun scope-subcheck-2 (setted-view view)
  ;;; one view equal interactive part, another equal functional part?
  (or (and (eql setted-view :interactive) (eql view :functional))
      (and (eql setted-view :functional) (eql view :interactive))))

(defun scope-subcheck-3 (setted-view view)
  ;;; one view equal decomposing part, another equal decomposing part?
  (or (and (eql setted-view :decomposing) (eql view :superposing))
      (and (eql setted-view :superposing) (eql view :decomposing))))
```

```
;;; Push variable to control flavor instance.

(defmacro push-event-variable (var)
  `(push-variable *variables* ,var :event))

(defmacro push-status-variable (var)
  `(push-variable *variables* ,var :status))

(defmacro push-adress-variable (var)
  `(push-variable *variables* ,var :adress))

(defmacro push-parameter-variable (var)
  `(push-variable *variables* ,var :parameter))

(defmacro push-constant (const)
  `(push-variables *variables* ,const :constant))

(defmethod (push-variable VARIABLES)(var key)
  (case key
    (:event     (push var event-variables))
    (:status    (push var status-variables))
    (:adress    (push var adress-variables))
    (:parameter (push var parameter-variables))
    (:constant  (push var constant))))

(defmethod (push-event-part EVENT-VARIABLE)(new-part)
  (unless (member new-part part)
    (push new-part part)))

(defmethod (push-view-value VIEW-VARIABLE)(push-value push-view)
  (push push-view view)
  (push `(,push-value) value))


;;; Change value

(defmethod (change-view-value VIEW-VARIABLE)(change-value change-view)
  (setq view `(,change-view))
  (setq value `((,change-value))))

(defmethod (change-view-value-2 VIEW-VARIABLE)(change-view change-value)
  (setq view  change-view)
  (setq value change-value))

(defun change-part (part)
  (case part
    (:decomposing :@dcmp)
    (:functional  :@func)
    (:superposing :@supr)))

(defun change-part2 (part)
  (case part
    ((:@dcmp dcmp :decomposing) :decomposing)
    ((:@func func :functional)  :functional)
    ((:@supr supr :superposing) :superposing)
    (otherwise      part)))
```

```
;;; Some Get functions

;;; Get values

(defun get-view-point (view)
  (case view
    (:global 5)
    (:interactive 4)
    (:functional 4)
    (:decomposing 3)
    (:superposing 3)
    (t (error-call 8 view))))

(defun get-status-value (variable-name part)
  (without-interrupts
    (get-view-variable-value (symbol-value variable-name) part)))

(defun get-address-value (variable-name part)
  (without-interrupts
    (get-view-variable-value (symbol-value variable-name) part)))

(defun get-constant-value (constant-name part)
  (without-interrupts
    (get-view-variable-value (symbol-value constant-name) part)))

(defmethod (get-view-variable-value VIEW-VARIABLE)(part)
  (let ((pos 0))
    (if (eql value :unbound)
        value
        (case (length view)
          (1 (if part (and (scope-check-internal view part) (car value)) (car value)))
          (t (if (setq pos (position part view))
                 (nth pos value)
                 (get-view-variable-value-1 self part)))))))

(defmethod (get-view-variable-value-1 VIEW-VARIABLE)(part)
  (loop for scope in view
        for i from 0
        when (scope-check-internal '(,scope) part)
          do (return (nth i value))))

(defun get-parameter-value (variable-name)
  (without-interrupts
    (get-parameter-value-with-instance (symbol-value variable-name))))

(defmethod (get-parameter-value-with-instance PARAMETER-VARIABLE)()
  value)


(defun get-part (part)
  (case part
    ((:@dcmp :decomposing) 'dcmp)
    ((:@func :functional)  'func)
    ((:@supr :superposing) 'supr)
    (:@timer               'timer)
    (otherwise             'unknown)))


(defmethod (get-timer-pos EVENT-VARIABLE) (to)  ;ogata
  (loop for val in value
        for i from 0
        when (eql (first val) to)
          do (return i)
          finally (return nil)))
```

```
;;; Separate action and timing parameter

(defun separate-action (action)
  (let* ((timer1 (second (member :timing action)))
         (timer2 (and timer1 (car (expand-variables `(,timer1))))))
    (when timer1
      (setq action (remove :timing action))
      (setq action (remove timer1 action)))
    (values action timer2)))


;;; Expand variables

(defun expand-variables (vars)
  (let (return-list)
    (dolist (var vars return-list)
      (if var
          (setq return-list `(,@return-list ,@(expand-variable var)))
          (tail-push return-list var)))))
;  (loop for var in vars
;        append (expand-variable var))) ;;; collect?

(defun expand-variable (var)
  (cond ((eql var :@self) `(,*self-process-id*))
        ((send-self-process-p var) `(,var))
        ((listp var)               `(,(expand-variables var)))
        (t (case (variable-type var)
             (:status    (get-status-value var *part*))
             (:adress    (get-adress-value var *part*))
             (:parameter (get-parameter-value var))
             (:constant  (get-constant-value var *part*))
             (t          `(,var))))))

(defun expand-parameter-list (parameter-list)
  (let (return-list)
    (dolist (para parameter-list return-list)
      (if para
          (setq return-list `(,@return-list ,@(expand-parameter-list-internal para)))
          (tail-push return-list para)))))
;  (loop for para in parameter-list
;        append (expand-parameter-list-internal para))) ;;; collect?

(defun expand-parameter-list-internal (para)
  (let (value)
    (if (and (eql (variable-type para) :parameter)
             (eql (get-parameter-value para) :unbound))
        `(,para)
        (if (listp (setq value (expand-variable para))) value `(,value)))))
```

```
;;; Reset functions

(defun reset-stacked-event-variables (variables part)
  (dolist (variable variables)
    (reset-stacked-event-variable variable part)))

(defun reset-stacked-event-variable (condition part)
  (if (eql (car condition) 'timeout)
      (reset-stacked-timeout-event-variable condition)
      (reset-stacked-event-variable-1 condition part)))

(defun reset-stacked-timeout-event-variable (condition)
  (multiple-value-bind (variable from value)
      (values (car condition) (second condition) (cddr condition))
    (setq value (expand-variables value))
;    (setq from (car (expand-variables '(,from))))
    (reset-timeout-event-variable-internal (symbol-value variable) from value)))

(defun reset-stacked-event-variable-1 (condition part)
  (multiple-value-bind (variable from value)
      (values (car condition) (second condition) (cddr condition))
    (setq value (expand-variables value))
    (setq from (car (expand-variables '(,from))))
    (reset-event-variable-internal (symbol-value variable) part from value)))

(defmethod (reset-variables-slots VARIABLES)()
  (setq event-variables     nil
        adress-variables    nil
        status-variables    nil
        parameter-variables nil
        constant            nil))

;;; following function reset TIMEOUT event

(defmethod (reset-timeout-event-variable-internal EVENT-VARIABLE)(from-part cut-value)
  from-part
  (let ((pos (get-timer-pos self (car (expand-variables cut-value)))))
    (when pos
      (setq from (nth-remove pos from)
            value (nth-remove pos value))
      (and (not from) (setq from :unbound))
      (and (not value) (setq value :unbound)))))
;  (let* ((pos (position from-part from :test #'pos-test))
;         (cut-value (if pos (nth pos value) cut-value)))
;    (when pos
;      (setq from (remove from-part from :test #'equal)
;            value (remove cut-value value :test #'equal))
;      (and (not from) (setq from :unbound))
;      (and (not value) (setq value :unbound)))))

;;; for another event

(defmethod (reset-event-variable-internal EVENT-VARIABLE)(part &optional from-part cut-value)
  (if (and from-part (eql part :superposing))
      (if cut-value
          (let* ((pos-list (get-supr-cut-value cut-value value))
                 (pos (position from-part from :test #'pos-test))
                 (pos (member pos pos-list)))
            (when pos
              (setq pos (car pos))
              (setq from (nth-remove pos from)  ;;(remove from-part from :test #'equal)
                    value (nth-remove pos value)) ;;(remove cut-value value :test #'equal))
              (and (not from) (setq from :unbound))
              (and (not value) (setq value :unbound))))
          (let* ((pos (position from-part from :test #'pos-test))
                 (cut-value (if pos (nth pos value) cut-value))) cut-value
                (when pos
                  (setq from (nth-remove pos from)  ;;(remove from-part from :test #'equal)
                        value (nth-remove pos value)) ;;(remove cut-value value :test #'equal
))
                  (and (not from) (setq from :unbound))
                  (and (not value) (setq value :unbound)))))
      (setq from :unbound)
      (setq value :unbound)))

;;
```

```
(defun get-supr-cut-value (cut-value values)
  (do ((pos 0 (1+ pos))
       (value values (cdr value))
       (pos-list nil))
      ((null value) pos-list)
    (when (parameter-matching (car value) cut-value)
      (push pos pos-list))))

(defun nth-remove (pos list)
  (do ((i 0 (1+ i))
       (j list (cdr j))
       (k nil))
      (nil)
    (cond ((= pos i) (return `(,@(reverse k) ,@(cdr j))))
          ((> i pos) (return `(,@(reverse k)))))
    (push (car j) k)))

;;

(defun reset-stacked-parameter-variables (variables)
  (dolist (variable variables)
    (reset-parameter-variable (symbol-value variable))))

(defun reset-parameter-variables ()
  (without-interrupts
    (reset-parameter-variables-internal *variables*)))

(defmethod (reset-parameter-variables-internal VARIABLES)()
  (dolist (var parameter-variables)
    (reset-parameter-variable (symbol-value var))))

(defmethod (reset-parameter-variable PARAMETER-VARIABLE)()
  (setq value :unbound))

(defun reset-all-decomposing-event-variables ()
  (without-interrupts
    (reset-all-event-variables *variables* :decomposing)))

(defun reset-all-functional-event-variables ()
  (without-interrupts
    (reset-all-event-variables *variables* :functional)))

(defun reset-all-superposing-event-variables ()
  (without-interrupts
    (reset-all-event-variables *variables* :superposing)))

(defmethod (reset-all-event-variables VARIABLES)(part)
  (dolist (var event-variables)
    (and (used-in-superposing var part)
         (reset-event-variable-internal (symbol-value var) part))))

(defun used-in-superposing (var part)
  (member part (EVENT-VARIABLE-part (symbol-value var))))


;;; Reset buffers and id

(defun reset-buffers-and-id-in-process-1 (flg)
  (setq *standard-output-flag* flg
        *round-id* 0
        *execution-list* nil
        *output-que* nil
        *event-to-functional* nil
        *event-to-superposing* nil))

(defun reset-buffers-and-id-in-process-2 ()
  (setq *return-process-id* nil
        *return-round-id* nil
        *return-process* nil))

(defun reset-make-instance-buffer-flag ()
  (setq *error* nil
        *constraint* nil
        *read-error-buffer* nil))

(defun reset-read-csl-buffer ()
  (setq *decomposing-part*  nil
        *functional-part*   nil
```

```
        *superposing-part*  nil
        *constant-variable* nil))

;;; Reset execution-list

(defun reset-execution-list ()
  (setq *execution-list* nil))

;;; Remake execution-list

(defun remake-execution-list (to-id)
  (let ((execution (find to-id *execution-list* :key #'second)))
    (and execution (setq *execution-list* (remove execution *execution-list* :test #'equal)))
))
```

```
;;; Set value to variable

(defun set-functional-event ()
  (dolist (event-list *event-to-functional*)
    (set-self-process-event
      (first event-list)
      (second event-list)
      (third event-list)
      :@func)))

(defun set-superposing-event ()
  (dolist (event-list *event-to-superposing*)
    (set-self-process-event
      (first event-list)
      (second event-list)
      (third event-list)
      :@supr)))

;;;

(defmethod (set-self-process-event EVENT-VARIABLE) (from-part set-value to-part)
  (case to-part
    (:@func (setq from from-part value set-value))
    (:@supr (and (eql from :unbound) (setq from nil))
            (and (eql value :unbound) (setq value nil))
            (push from-part from)
            (or set-value (setq set-value '(,set-value)))
            (push set-value value))))


(defun set-event-value (event to-part from-part set-value &optional (timer 0))
  (let ((que-list (make-output-que-list event *self-process-id* to-part
                                        *self-process-id* from-part timer set-value)))
    (push-output-que que-list)
    (when (and (eql to-part :@intr) (event-variable-existp event))
      (setq to-part :@supr))
    (when (event-variable-existp event)
      (without-interrupts
        (set-event-value-internal (symbol-value event) to-part from-part set-value)))))

(defmethod (set-event-value-internal EVENT-VARIABLE)(to-part from-part set-value)
  (and (eql to-part :@self) (setq to-part from-part from-part :@self))
  (when (member (change-part2 to-part) part)
    (if (and (self-process-part-p from-part)
             (not (eql from-part to-part)))
        (set-event-value-self-process self to-part from-part set-value)
        (case to-part
          ((:@dcmp :@func) (setq from from-part value set-value) T)
          (:@supr          (and (eql from :unbound) (setq from nil))
                           (and (eql value :unbound) (setq value nil))
                           (push from-part from)
                           (or set-value (setq set-value '(,set-value)))
                           (push set-value value)
                           T)))))

(defmethod (set-event-value-self-process EVENT-VARIABLE)(to-part from-part set-value)
  (when (and (eql from-part :@supr) (eql to-part :@func))
    (setq *csl-flow* :functional))
  (case  to-part
    (:@dcmp)
    (:@func (tail-push *event-to-functional* '(,self ,from-part ,set-value)) T)
    (:@supr (tail-push *event-to-superposing* '(,self ,from-part ,set-value)) T)))

(defun set-status-value (status set-value part)
  (without-interrupts
    (set-view-variable-value (symbol-value status) set-value part)))

(defun set-adress-value (adress set-value part)
  (without-interrupts
    (set-view-variable-value (symbol-value adress) set-value part)))

(defun set-constant-value (const set-value part)
  (without-interrupts
    (set-view-variable-value (symbol-value const) set-value part)))

(defmethod (set-view-variable-value VIEW-VARIABLE)(set-value part)
  (let ((pos 0))
```

```
    (case (length view)
      (1 (and (scope-check-internal view part) (setq value '(,set-value)))) ;;; really ok?
      (t (if (setq pos (position part view))
             (setq value (substitute set-value (nth pos value) value :test #'equal :start pos
 :count 1))
             (set-view-variable-value-1 self set-value part))))))

(defmethod (set-view-variable-value-1 VIEW-VARIABLE)(set-value part)
  (let ((flg nil))
    (dolist (scope view flg)
      (and (scope-check-internal '(,scope) part)
           (setq value (substitute set-value (nth (position scope view) value) value :test #'
equal :start (position scope view) :count 1))
           (setq flg T)))))

(defun set-parameter-value (parameter set-value)
  (without-interrupts
    (set-parameter-value-internal (symbol-value parameter) set-value)))

(defmethod (set-parameter-value-internal PARAMETER-VARIABLE)(set-value)
  (setq value set-value))


;;; Send EVENT another process and set value to variable.

(defun set-and-send-event-variable (variable to from value timer)
  (incf *round-id*)
  (let ((que-list (make-output-que-list variable to 'unknown *self-process-id* from timer val
ue)))
    (push-output-que que-list)
    (when (boundp variable)
      (without-interrupts
        (set-event-value-internal (symbol-value variable) to from value)))))


;;;

(defun set-input-event (event-name to to-part from from-part
                        process-id round-id timer &rest value)
  timer to from-part
  (when (event-variable-existp event-name)
    (cond ((eql *device-id* from) (setq from :@dev to-part :@dcmp))
          (t (setq *return-process-id* process-id
                   *return-round-id* round-id
                   to-part :@dcmp
                   *return-process* from)))
    (without-interrupts
      (set-event-value-internal (symbol-value event-name) to-part from value))))

;;;

(defun get-variable-pair-list (condition)
  (get-variable-pair-list-internal (loop for cond in condition
                                         for (event-var from-part) = cond
                                         when (member from-part '(:@intr :@supr))
                                           collect (get-one-event-variable-pair-list (symbol-
value event-var)
                                                                                     event-va
r))))

(defun get-variable-pair-list-internal (list1)
  (loop for i from 0
        for list2 = (loop for x in list1
                          for leng = (length x)
                          when (< i leng)
                            collect (nth i x))
        unless (loop for x in list2
                     when x do (return nil)
                     finally (return t))
          collect list2 into list3
        else do (return list3)))

(defmethod (get-one-event-variable-pair-list EVENT-VARIABLE)(event-name)
  (when (and (not (eql from :unbound)) (not (eql value :unbound)))
    (loop for from-part in from
          for val in value
          collect '(,event-name ,from-part ,val))))
```

```
(defun preset-event-variable-value (variable-pair-list)
  (dolist (variable-pair variable-pair-list)
    (preset-event-variable-value-internal (symbol-value (car variable-pair))
                                          (second variable-pair)
                                          (third variable-pair))))

(defmethod (preset-event-variable-value-internal EVENT-VARIABLE)(from-part preset-value)
  (setq from `(,from-part)
        value `(,preset-value)))

(defun episet-event-variable-value (variable-pair-list)
  (dolist (variable-pair variable-pair-list)
    (dolist (var-pair variable-pair)
      (event-variable-value-nil-set (symbol-value (car var-pair)))))
  (dolist (variable-pair variable-pair-list)
    (dolist (var-pair variable-pair)
      (episet-event-variable-value-internal (symbol-value (car var-pair))
                                            (second var-pair)
                                            (third var-pair)))))

(defmethod (episet-event-variable-value-internal EVENT-VARIABLE)(from-part set-value)
  (push from-part from)
  (push set-value value))

(defmethod (event-variable-value-nil-set EVENT-VARIABLE)()
  (setq from  nil
        value nil))
```

```
;;; Make output EVENT list

(defun make-output-que-list (event-name to to-part from from-part timer values)
  (let ((to-part (get-part to-part))
        (from-part (get-part from-part)))
    (if (eql to *return-process*)
        `(EVENT ,event-name ,to ,to-part ,from ,from-part ,*return-process-id*
                ,*return-round-id* ,timer ,@values)
        `(EVENT ,event-name ,to ,to-part ,from ,from-part ,*self-process-id*
                ,*round-id* ,timer ,@values))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Global variables for communication.
;;;

;;; Some buffers

(defvar *input-buffer* nil)

(defvar *output-buffer* nil)


;;; Input que

(defvar *input-que*  nil)


;;; Output que

(defvar *output-que* nil)
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Input EVENT processing.
;;;

(defun input-process (self-process-id)
  (reset-buffers-and-id-in-input)
  (setq *self-process-id* self-process-id)
  (input-process-internal))

(defun input-process-internal ()
  (loop do
    (when *step-exec-in*
      (input-process-exec)
      (and *suspend-flag* (setq *step-exec-in* nil)))))

(defun input-process-exec ()
  (loop for input-event = nil
        for input-flag  = nil
    do
    (when *input-buffer*
      (without-interrupts
        (setq input-event (cdr (pop *input-buffer*))))  ;;; ?
      (case (setq input-flag (apply #'input-event-check input-event))
        (1 (push-input-que input-event))
        (2 (apply #'set-event-value-from-input input-event))
        (3 (apply #'timeout-event-happen input-event)))
      (if (eql input-flag 2) (return nil)))
    (when (not *input-buffer*) (return nil))
    ))

(defun input-event-check (event-name to to-part from from-part
                          process-id round-id timer &rest values)
  ;;; return value 1 ==> push input que
  ;;; return value 2 ==> set event value to instance
  ;;; return value 3 ==> Timeout event
  ;;;                    Now not enter Timeout event. if entered, I ignore this event.  7/12/
90
  ;;; return value 4 ==> Timeout happen or ignore input event.
  event-name to to-part from from-part timer values
  (if (eql *self-process-id* process-id)
      (input-event-check-internal event-name process-id round-id)
      1))

(defun input-event-check-internal (event-name process-id round-id)
  (let ((id '(,process-id ,round-id))
        flg)
    (cond ((eql event-name 'Timeout) 4)
          (*execution-list* (dolist (event *execution-list* (if flg 4 1))
                              (cond ((equal id (car event))
                                     (return 2))
                                    ((eql process-id (caar event)) (setq flg t)))))
          (t 4))))

;;; Timeout event

(defun timeout-event-happen (event-name to to-part from from-part
                             process-id round-id timer &rest values)
  event-name to to-part from-part timer values
  (when (boundp 'timeout)
    (remove-execution-list process-id round-id from)
    (without-interrupts
      (set-event-value-internal (symbol-value 'timeout) :@supr from '(,process-id ,round-id))
)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Output EVENT processing
;;;

(defun output-process ()
  (reset-buffers-and-id-in-out)
  (output-process-internal))

(defun output-process-internal ()
  (loop for output-event = nil
        do
    (when *output-que*
      (without-interrupts
        (setq output-event (pop *output-que*)))
      (case (output-event-check output-event)
        (1 (send-event-any-process output-event))
        (2 (send-event-another-process output-event :timer))
        (3 (send-command-from-output-que output-event))))
    ))

(defun send-event-another-process (output-event &optional timer)
  (if (eql (third output-event) *self-process-id*)
      (send-event-self-process output-event)
      (send-event-out-process output-event))
  (and timer (timer-process-start output-event)))

(defun send-event-out-process (output-event)
  (let* ((round-id (eighth output-event))
         (execution `((,*self-process-id* ,round-id) ,(third output-event)))
         (output-event (cons (get-stream-from-host *displayer-name*) output-event))
         )
    (without-interrupts
      (push execution *execution-list*))
    (push-output-buffer output-event)))

(defun send-event-self-process (output-event)
  (let ((output-event (cons (get-stream-from-host *displayer-name*) output-event)))
    (push-output-buffer output-event)))

(defun send-event-any-process (output-event)
  (send-event-another-process output-event))

(defun send-command-from-output-que (output-command)
  (let ((output-command (cons (get-stream-from-host *displayer-name*) output-command)))
    (push-output-buffer output-command)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Utility functions for Communication module.
;;;

;;; Some utility functions

(defun string-to-symbol (str)
  (read-from-string str))

(defun symbol-to-string (sym)
  (format nil "~A" sym))


(defmacro tail-push (buffer item)
  `(setq ,buffer (append ,buffer (list ,item))))


(defun output-event-check (output-event)
  (cond ((eql (car output-event) 'event)
         (when (>= (length output-event) 9)
           (if (= (ninth output-event) 0) 1 2)))
        (T 3)))

;;; Change input EVENT list to symbols list

(defun change-input-event-list (string-list)
  (loop for str in string-list
        collect (string-to-symbol str)))


;;; Change output EVENT list to string list

(defun change-output-event-list (symbol-list)
  (loop for sym in symbol-list
        collect (symbol-to-string sym)))


;;; Reset function

(defun reset-buffers-and-id-in-input ()
  (setq *self-process-id* nil
        *input-que*       nil))

(defun reset-buffers-and-id-in-out ()
  (setq *output-buffer* nil))

(defun remove-execution-list (process-id round-id from)
  (let ((execution '((,process-id ,round-id) ,from)))
    (setq *execution-list* (remove execution *execution-list* :test #'equal))))


;;; Set event variable

(defun set-event-value-from-input (event-name to to-part from from-part
                                   process-id round-id timer &rest value)
  to to-part from-part timer
  (remove-execution-list process-id round-id from)
  (without-interrupts
    (set-event-value-internal (symbol-value event-name) :@supr from value)))


;;; Push event-list to buffer.

(defun push-input-buffer (input-event)
  (without-interrupts
    (tail-push *input-buffer* input-event)))

(defun push-input-que (input-event)
  (without-interrupts
    (tail-push *input-que* input-event)))

(defun push-output-buffer (output-event)
  (without-interrupts
    (tail-push *output-buffer* output-event)))

(defun push-output-que (output-event)
```

```
  (without-interrupts
    (tail-push *output-que* output-event)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-


;;;
;;; Timer process's Global variables
;;;

;;; Timer control list
(defvar *timer-list* nil)
```

```
;;; -*- Mode: LISP; Base: 10; Syntax: Common-Lisp; Package: CSL -*-

;;;
;;; timer function
;;;


;;; timer utillity

(defun push-timer-list (list)
  (without-interrupts
    (push list *timer-list* )))

(defun timer-search-remove (to-id)
  (loop for one in *timer-list*
        when (eql to-id (third (car one)))
          do (setq *timer-list* (remove one *timer-list* :test #'equal)) (return one)))

(defun kill-timer-list ()
  (dolist (timer *timer-list*)
    (send (second timer) :kill))
  (setq *timer-list* nil))


;;; timer main function
;;; start

(defun timer-process-start (arg)
  (let ((timer-id))
    (when (< 0 (ninth arg))
      (setq timer-id (process-run-function 'timer #'run-timer  arg))
      (push-timer-list (list arg  timer-id)))))

;;; sleep

(defun run-timer (arg)
  (sleep (/ (ninth arg) 10.0))
  (apply #'timer-process-end arg))

;;; end

(defun timer-process-end (event event-name to to-part from from-part r-p-id r-r-id &rest para
)
  event event-name to-part from-part para from
  (when (timer-search-remove to)
    (remake-execution-list to)
    (set-event-value 'TIMEOUT :@supr :@timer '(,to ,r-p-id ,r-r-id))))
;    (push-input-buffer '(EVENT Timeout ,from ,from-part ,to ,to-part ,r-p-id ,r-r-id ,para))
;    (push-output-que '(EVENT Timeout ,from unknown device unknown ,r-p-id ,r-r-id 0 ,to))))

;;; Release timer

(defun reset-timer (to-id)
  (let ((timer-id (timer-search-remove to-id)))
    (and timer-id (send (cadr timer-id) :kill))))
```

```
;;; -*- Mode: LISP; Base: 10; Syntax: Common-Lisp; Package: CSL -*-

;;;
;;; flavor definition
;;;

;; for Network

(defflavor CSL-mixin
        (stream)
        ()
  :abstract-flavor
  (:initable-instance-variables stream)
  (:required-init-keywords :stream) )

(defflavor CSL-SERVER
        ()
        (CSL-mixin) )

(defflavor CSL-CLIENT
        ()
        (CSL-mixin) )

;; manage network-table

(defflavor NW-MANAGER
        ((process-id     nil)
         (load-host      nil)
         (target-spec    nil)
         (access-stream  nil)
         (service-status nil))
        ()
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)
```

```
;;; -*- Mode: LISP; Base: 10; Syntax: Common-Lisp; Package: CSL -*-

;;;
;;; global variables for Network-I/O
;;;


(defvar *host-name* nil)            ; for client-start

(defvar *server-part* nil)          ; 表示系(:display) or 処理系(:practice)

(defvar *csl-system-configuration-table* nil)       ; (nw-manager-instance1  ... )

;;
;; for 処理系
;;

(defvar *file-name* nil)            ; spec-file name
(defvar *file-list* nil)            ; spec-file list
(defvar *read-file-list* nil)

(defvar *displayer-name* nil)       ; hyouji-kei host-name
(defvar *service-status* nil)       ; nil -> ready -> enable

;; process-id of 入力部  出力部 ルール実行部
(defvar *in-process* nil)
(defvar *out-process* nil)
(defvar *pro* nil)
(defvar *client-of-practice* nil)


;;
;; for 表示系
;;

(defvar *display-process-id* nil)
(defvar *client-of-display*  nil)


;;
;; for debugger
;;

(defvar *execution-mode* nil)
(defvar *change-part* nil)
(defconstant *execution-mode-list* '(normal debug))
(defconstant *part-list* '(decomposing functional superposing))

(defvar *suspend-flag* nil)
(defvar *step-exec-in* nil)
(defvar *step-exec-pro* nil)
```

```
;;; -*- Mode: LISP; Base: 10; Syntax: Common-Lisp; Package: CSL -*-

;;;
;;; for Server (network-server)
;;;

(net:define-server :csl-protocol
    (:medium :byte-stream :stream stream)
   (let ((server (make-instance 'CSL-SERVER :stream stream)))          ; サーバーインスタンス作成
     (server-init)
     (loop  for command = (read-client-command server)                ; 読み込みループ
            for cont-flag = (do-command command )
            unless cont-flag
              do (loop-finish) )))

(defmethod (read-client-command CSL-SERVER) ()                        ; ストリームから読み込み
  (let ((command (send stream :line-in)))                             ; SUCCESS or ERROR
    (cond ((< 0 (string-length command))                              ; を返答する
           (format stream "SUCCESS~%")
           (send stream :force-output)
           (read-from-string command))
          (T (format stream "ERROR~%")
             (send stream :force-output)
             nil ))))

(defun do-command (command)
  (if (equal "EVENT" (car command))
      (send-event command)
      (init-process command) ))


; push event *input-buffer*

(defun send-event (command)
  (push-input-buffer (change-command-to-symbol command)) )
```

```
;;
;; init-process 各コマンド毎処理を行なう
;;
(defun init-process (command)
  (let ((cont-flag T))
    (selector (car command) string-equal
      (("LOAD-CONFIGURATION") (load-configuration-file (second command)))
      (("END")            (end-command) (setq cont-flag nil))

  ;; from hyouji-kei to syori-kei.
      (("SERVER-HOST")      (server-host command))
      (("LOAD-SPEC")        (load-spec command))
      (("APPEND-LOAD-SPEC") (append-load-spec command))
      (("INIT-SYSTEM")      (init-system (intern (second command) 'csl)))

  ;; from syori-kei to hyouji-kei.
      (("CLIENT-READY")     (client-ready command))
      (("SERVICE-ENABLE")   (service-enable command))
      (("READ-ERROR")       (read-error command))

  ;; for debugger
  ;; from hyouji-kei to syori-kei.
      (("SET-MODE")             (set-mode command))
      (("SHOW-VARIABLES-VALUE") (show-variables-value command))
      (("SHOW-CONSTRAINTS")     (show-constraints command))
      (("SHOW-EXEC-PART")       (show-exec-part command))
      (("SET-VARIABLES-VALUE")  (set-variables-value command))
      (("SUSPEND-EVENT-TRANSACTION") (suspend-event-transaction command))
      (("RESUME-EVENT-TRANSACTION")  (resume-event-transaction command))
      (("INVOKE-NEXT-EVENT-TRANSACTION")(invoke-next-event-transaction command))
  ;; from syori-kei to hyouji-kei.
      (("FIRED-RULE")        (fired-rule command))
      (("CURRENT-EXEC-PART") (current-exec-part command))
      (("VARIABLES-VALUE")   (variables-value command))
      (("CONSTRAINTS")       (constraints command))
      (("RESULT-SET-VARIABLES-VALUE") (result-set-variables-value command)))
    cont-flag ))                                          ; only end-command return nil
                                                          ; otherwise        return T

; command which syori-kei & hyouji-kei receive

(defun end-command ()
  (setq *service-status* nil)
  (when (eql *server-part* :practice)
    (server-push-command *displayer-name* '(end))))

; command which syori-kei only receive

(defun server-host (command)
  (setq *server-part* :practice
        *csl-system-configuration-table* nil
        *output-buffer* nil
        *displayer-name* (car (change-command-to-symbol (cdr command))))
  (kill-process)
  (make-and-set-nw-instance :load-host *displayer-name*)
  (setq *client-of-practice* (process-run-function 'send-client #'send-client-command) )
  (server-push-command *displayer-name*
                       '(client-ready ,(send net:*local-host* :name)) )
  (setq *service-status* :ready) )


(defun load-spec (command)
  (let ((file-list (read-from-string (third command))))
    (when (and (eql *server-part* :practice)
               (and file-list (listp file-list))
               *service-status* )
      (setq *file-list* nil
            *self-process-id* (car (change-command-to-symbol '(,(second command)))))
      (load-spec-internal *self-process-id* file-list))))

(defun load-spec-internal (id file-list)
  (setq *output-buffer* nil)
  (kill-process T)
  (make-variables-instance file-list)
  (if *error*
      (read-spec-error)
      (main-process-start id) ))
```

```
(defun append-load-spec (command)
  (let ((id (car (change-command-to-symbol '(,(second command)))))
        (file-list (read-from-string (third command))))
    (when (and (eql *server-part* :practice)
               (eql *self-process-id* id)
               (eql *service-status* :enable)
               (and file-list (listp file-list))
               *file-list*      )                          ;??
      (append-load-spec-internal id file-list))))

(defun append-load-spec-internal (id file-list)
  (setq *output-buffer* nil)
  (kill-process T)
  (make-variables-instance file-list T)
  (if *error*
      (read-spec-error)
      (main-process-start id) ))
```

```
(defun init-system (id)
  (setq *output-buffer* nil)
  (when (and (eql *server-part* :practice)
             (eql *self-process-id* id)
             *file-list*)
    (kill-process T)
    (make-variables-instance-internal)
    (if *error*
      (read-spec-error)
      (main-process-start id) )))

(defun main-process-start (id)
  (reset-debugger-flag)
  (setq *input-buffer* nil
        *in-process* (process-run-function 'in #'input-process id)
        *out-process* (process-run-function 'out #'output-process)
        *pro* (process-run-function 'pro #'process-start ) )
  (setq *service-status* :enable)
  (server-push-command *displayer-name* '(service-enable ,*self-process-id*)))


;;; Read spec error
;;;   Read error sending S-expression is (READ-ERROR process-id ("file-name"..) "error reason
")

(defun read-spec-error ()
  (let* ((instance (get-instance :host *host-name*))
         (stream (get-info instance :stream)))
    (loop for error in *read-error-buffer*
          do
      (push-output-buffer
        '(,stream read-error ,*self-process-id* ,(make-erase-cr-code *file-list*) ,error))
      (sleep 0.1)))
  (setq *service-status* :ready))


; command which hyouji-kei only receive

(defun client-ready (command)
  (let* ((host-name (car (change-command-to-symbol (cdr command))))
         (host-name (if (stringp host-name) (read-from-string host-name) host-name))
         (instance (get-instance :host host-name)))
    (table-info-change instance :service-status :ready)
    (make-and-send-command instance :load-spec) ))

(defun service-enable (command)
  (let* ((process-id (car (change-command-to-symbol (cdr command))))
         (instance (get-instance :id process-id)))
    (table-info-change instance :service-status :enable) ))

(defun read-error (command)
  (let* ((process-id (car (change-command-to-symbol (cdr command))))
         (instance (get-instance :id process-id)))
    (format t "    Process-id ~a     File-list ~a ~%    Error reason ~s~%~%"
            process-id (read-from-string (third command)) (fourth command))
    (table-info-change instance :service-status :ready)))
```

```
;;; Debugger command
;;  debugger command which syori-kei only receive

(defun set-mode (command)
  (let* ((com (change-command-to-symbol (cdr command)))
         (id   (first com))
         (mode (second com)))
    (when (and (command-receive-ok id)
               (member mode *execution-mode-list*))
      (setq *execution-mode* mode)
      (if (eql mode 'debug) (send-current-exec-part *change-part*)))))

(defun show-variables-value (command)
    (when (command-receive-ok (car (change-command-to-symbol (cdr command))))
      (let ((variables-list (make-variables-value-list)))
        (send-variables-value-list variables-list))))

(defun show-constraints (command)
  (let* ((com (change-command-to-symbol (cdr command)))
         (id   (first com))
         (part (second com)))
    (when (and (command-receive-ok id)
               (member part *part-list*))
      (send-constraints part))))

(defun show-exec-part (command)
  (when (command-receive-ok (car (change-command-to-symbol (cdr command))))
    (send-current-exec-part *change-part*)))

(defun suspend-event-transaction (command)
  (when (command-receive-ok (car (change-command-to-symbol (cdr command))))
    (setq *suspend-flag*  T
          *step-exec-in*  nil
          *step-exec-pro* nil )))

(defun resume-event-transaction (command)
  (when (command-receive-ok-2 (car (change-command-to-symbol (cdr command))))
    (setq *suspend-flag*  nil
          *step-exec-in*  T
          *step-exec-pro* T )))

(defun invoke-next-event-transaction (command)
  (when (command-receive-ok-2 (car (change-command-to-symbol (cdr command))))
    (if (eql *change-part*  :idling)
        (if *input-que*
            (setq *step-exec-pro* T)
            (setq *step-exec-in*  T
                  *step-exec-pro* T))
        (setq *step-exec-in* T))))

(defun set-variables-value (command)
  (let* ((com (change-command-to-symbol (cdr command)))
         (id   (first com))
         (variables-list (second com))
         return-list)
    (when (command-receive-ok id)
      (setq return-list (set-variables-value-internal variables-list))
      (send-result-set-variables-value return-list))))

;;  debugger command which hyouji-kei only receive

(defun fired-rule (command)
  (let* ((com  (change-command-to-symbol (cdr command)))
         (id   (first com))
         (rule (second com))
         (parameter (third com)))
    (fired-rule-internal id rule parameter)))

(defun current-exec-part (command)
  (let* ((com  (change-command-to-symbol (cdr command)))
         (id   (first com))
         (part (second com)))
    (current-exec-part-internal id part)))

(defun variables-value (command)
  (let* ((com  (change-command-to-symbol (cdr command)))
         (id   (first com))
```

```
        (variables-list (second com)))
    (variables-value-internal id variables-list)))

(defun constraints (command)
  (let* ((com  (change-command-to-symbol (cdr command)))
         (id   (first com))
         (part (second com))
         (receive-constraints (third com)))
    (constraints-internal id part receive-constraints)))

(defun result-set-variables-value (command)
  (let* ((com  (change-command-to-symbol (cdr command)))
         (id   (first com))
         (variables-list (second com)))
    (result-set-variables-value-internal id variables-list)))
```

```
;;; for User (network-client)

(defun make-client-instance (host)                          ; client instance 作成
  (setq *host-name* host
        host (net:parse-host host))
  (net:invoke-service-on-host   :csl host) )

(net:define-protocol :csl-protocol
                     (:csl :byte-stream)
  (:invoke-with-stream
    (stream)
    (let ((client (make-instance 'CSL-CLIENT :stream stream)))
      (table-info-change (get-instance :host *host-name*)    ; *csl-sys...-table* 変更
                         :stream client) )))                 ; instance access-stream

(tcp:add-tcp-port-for-protocol :csl-protocol 321)

;;  Send-loop

(defun send-client-command()
  (loop do
    (when *output-buffer*
      (let* ((command (get-command-from-output-buffer))
             (to-instance (pop command)) )
        (send-command-internal to-instance command) ))))

(defmethod (send-command-internal CSL-CLIENT)(command)
  (loop do
    (let (answer)
      (format stream "(")
      (dolist (x command) ;;(change-command-to-string command))
        (format stream "~s " (format nil "~a" x)))
      (format stream ")~%")
      (send stream :force-output)               ; send command
      (setq answer (send stream :line-in))      ; receive ack
      (and (response-check answer) (loop-finish)) )))
```

```
;;; for Test

(defun test-top (file-name)                    ;line-mode
  (network-init)
  (client-init)
  (let (ret)
    (load-configuration-file file-name)
    (setq *server-part* :display
          *display-process-id* (process-run-function  'display #'display-start)
          *client-of-display* (process-run-function 'send-client #'send-client-command) )
    (send-all-practice :server-host)
    (when (practice-ready-p)
      (loop  do
        (setq ret (prompt-and-accept '(integer 1 12)
"~%Which Command?
     1.init-system
     2.event
     3.set-mode
     4.show-variables-value
     5.show-constraints
     6.append-load-spec
     7.suspend-event-transaction
     8.resume-event-transaction
     9.invoke-next-event-transaction
    10.set-variables-value
    11.show-exec-part
    12.end              =>"))
        (send-command-to-practice ret)
        (when (= ret 12)
          (sleep 3)
          (test-end)
          (loop-finish) )
        ))))

(defun test-end ()
  (all-client-stream-close)
  (kill-process-internal '(display send-client))
  (setq *display-process-id* nil
        *client-of-display*  nil ))


; 表示系起動
(defun display-start ()
  (loop when *input-buffer*
        do
          (let* ((command (get-command-from-input-buffer))
                 (to-id (third command))
                 (from-id (fifth command)))
            (format t "~%OUT==>~S~%" command)
            (unless (or (eql to-id from-id)
                        (eql to-id 'device)
                        (not (get-instance :id to-id)))
              (push-output-buffer
                (cons (get-info (get-instance :id to-id) :stream) command) )
              (format t "SEND OUT==>~S~%" command)))))
```

```
; send command 表示系 -> 処理系

(defun send-command-to-practice (num)
  (let (command to-id mode part spec var-list)
    (case num
      (1 (format t "~%Input which id  => ")     ; (init-system id)
         (setq to-id (read))
         (send-one-practice :init-system to-id))
      (2 (format t "~%Input EVENT Command  => ")        ; (event name to-id to-part .. )
         (setq command (read))
         (send-one-practice :event (third command) command))
      (3 (format t "~%Input which id  => ")     ; (set-mode id mode)
         (setq to-id (read))
         (setq mode (accept '((member normal debug)) :prompt "mode" :default 'normal))
         (send-set-mode to-id mode))
      (4 (format t "~%Input which id  => ")     ; (show-variables-value id)
         (setq to-id (read))
         (send-show-variables-value to-id))
      (5 (format t "~%Input which id  => ")     ; (show-constraints id part)
         (setq to-id (read))
         (setq part (accept '((member decomposing functional superposing)) :prompt "part" :de
fault nil))
         (send-show-constraints to-id part))
      (6 (format t "~%Input which id  => ")
         (setq to-id (read))
         (setq spec (prompt-and-accept '(list)
                                       "~%Input spec file list => "))
         (send-append-load-spec to-id spec))
      (7 (format t "~%Input which id  => ")     ; (suspend-event-transaction)
         (setq to-id (read))
         (send-suspend-event-transaction to-id))
      (8 (format t "~%Input which id  => ")     ; (resume-event-transaction)
         (setq to-id (read))
         (send-resume-event-transaction to-id))
      (9 (format t "~%Input which id  => ")     ; (invoke-next-event-transaction)
         (setq to-id (read))
         (send-invoke-next-event-transaction to-id))
      (10 (format t "~%Input which id  => ")    ; (set-variables-value)
         (setq to-id (read))
         (setq var-list (prompt-and-accept '(list)
                                       "~%Input var-view-value list => "))
         (send-set-variables-value to-id var-list))
      (11 (format t "~%Input which id  => ")    ; (show-exec-part)
         (setq to-id (read))
         (send-show-exec-part to-id))
      (12 (send-all-practice :end)) )))         ; (end)


(defun send-all-practice (key)                          ; send all 処理系
  (dolist (instance *csl-system-configuration-table*)
    (when (status-check instance key)
      (make-and-send-command instance key) )))

(defmethod (make-and-send-command NW-MANAGER) (key)
  ;;; illegal sending command
    (push-output-buffer
      (case key
        (:server-host `(,access-stream server-host ,(send net:*local-host* :name)))
        (:load-spec   `(,access-stream load-spec ,process-id ,target-spec))
        (:end         `(,access-stream end)) )))

(defun send-one-practice  (key id &optional command)    ;send one 処理系
  (let* ((instance (get-instance :id id))
         (stream (get-info instance :stream)))
    (and stream
         (status-check instance key)
         (push-output-buffer
           (case key
             (:init-system `(,stream init-system ,id))
             (:event
               (cons stream command)) )))))

(defun send-init-system (id)
  (push-command-output-buffer `(init-system ,id) id))

;(defun send-event (id command)
;  (push-command-output-buffer `(,command) id))
```

```
(defun send-set-mode (id mode)
  (when mode
    (push-command-output-buffer '(set-mode ,id ,mode) id)))

(defun send-show-variables-value (id)
  (push-command-output-buffer '(show-variables-value ,id) id))

(defun send-show-constraints (id part)
  (when part
    (push-command-output-buffer '(show-constraints ,id ,part) id)))

(defun send-append-load-spec (id spec-list)
  (push-command-output-buffer '(append-load-spec ,id ,spec-list) id))

(defun send-suspend-event-transaction (id)
  (push-command-output-buffer '(suspend-event-transaction ,id) id))

(defun send-resume-event-transaction (id)
  (push-command-output-buffer '(resume-event-transaction ,id) id))

(defun send-invoke-next-event-transaction (id)
  (push-command-output-buffer '(invoke-next-event-transaction ,id) id))

(defun send-set-variables-value (id var-list)
  (let ((var-list (change-command-to-string var-list)))
    (push-command-output-buffer '(set-variables-value ,id ,var-list) id)))

(defun send-show-exec-part (id)
  (push-command-output-buffer '(show-exec-part ,id) id))


; when end (stream close)

(defun all-client-stream-close()
  (dolist (one-data *csl-system-configuration-table*)
    (send-close one-data) )
  (setq *csl-system-configuration-table* nil) )

(defmethod (send-close NW-MANAGER)()
  (send-close-internal access-stream) )

(defmethod (send-close-internal CSL-CLIENT)()
  (send stream :close) )
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; Read Configuration file.
;;;

(defun load-configuration-file (file-name)
  (let ((file-name (make-configuration-file file-name)))
    (if (not file-name)
        (format t "The file ~A was not found.~%" file-name)
        (with-open-file (infile file-name :direction :input)
          (let ((comment (car (read infile nil :EOF))))
            (if (eql comment 'CSL-SYSTEM-Configuration)
                (do ((info (read infile nil :EOF) (read infile nil :EOF)))
                    ((or (eql info :EOF)
                         (and (not (eql 6 (length info)))
                              (not (format t "~%Check Configuration info length and try again
.~%")) )))
                  (unless (apply #'make-and-set-nw-instance info) (return)))
                (format t "~%Check Configuration comment and try again.~%")))
          (when *csl-system-configuration-table*
            (setq *server-part* :practice))))))

(defun make-and-set-nw-instance  (&key ((:process-id id)    nil)
                                       ((:load-host  host)  nil)
                                       ((:target-spec spec) nil))
  (when (listp spec)
    (push (make-instance 'NW-MANAGER
                         :process-id id :load-host host :target-spec spec)
          *csl-system-configuration-table* )
    (make-client-instance host)
    T))

(defun make-configuration-file (file-name)
  (let ((name (if (symbolp file-name) (string file-name) file-name)))
    (cond ((search ":>" name :test #'equal) name)
          ((search ";"  name :test #'equal) name)
          (T (format nil "csl:configuration;~a" file-name)))))
```

```
;;; -*- Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; DEBUGGER for Shori-kei
;;;

;; exec Variables-value command

(defun make-variables-value-list ()
  (let* ((adress-val-list (get-adress-valiables-list *variables*))
         (status-var-list (get-status-variables-list *variables*))
         (adress-var-value-list
           (make-variables-value-list-internal adress-val-list))
         (status-var-value-list
           (make-variables-value-list-internal status-var-list)))
    `(,@adress-var-value-list ,@status-var-value-list)))

(defun make-variables-value-list-internal (var-list)
  (let ((return-list nil))
    (loop for var in var-list
          do
      (setq return-list `(,@return-list ,@(var-view-value-list (symbol-value var) var))))
    return-list))

(defmethod (var-view-value-list VIEW-VARIABLE) (var-name)
  (let ((view-num (length view)))
    (cond ((eql view-num 1) `((,var-name ,(car view) ,(car value))))
          ((> view-num 1) (var-view-value-list-2 var-name view value)))))

(defun var-view-value-list-2 (var-name view value)
  (loop for view1 in view
        for value1 in value
        collect `(,var-name ,view1 ,value1) into var-list
        finally (return var-list)))


;; exec Set-variables-value command

(defun set-variables-value-internal (variables-list)
  (loop for variable-list in variables-list
        collect (set-variable-value variable-list) into return-list
        finally (return return-list)))

(defun set-variable-value (variable-list)
  (let ((variable (first  variable-list))
        (view     (change-command-view (second variable-list)))
        (value    (cddr variable-list)))
    (if (boundp variable)
        (case (variable-type (first variable-list))
          (:status (if (set-status-value variable value view)
                       (var-view-value-list-scope (symbol-value variable) variable view)
                       (set-variables-error 1 variable view)))
          (:adress (if (set-adress-value variable value view)
                       (var-view-value-list-scope (symbol-value variable) variable view)
                       (set-variables-error 1 variable view)))
          (otherwise   (set-variables-error 2 variable view)))
        (set-variables-error 3 variable view))))

(defmethod (var-view-value-list-scope VIEW-VARIABLE)(var-name part)
  (let ((pos 0))
    (case (length view)
      (1 (and (scope-check-internal view part) `(,var-name ,(car view) ,(car value))))
      (t (if (setq pos (position part view))
             `(,var-name ,(nth pos view) ,(nth pos value))
             (var-view-value-list-scope-1 self var-name part))))))

(defmethod (var-view-value-list-scope-1 VIEW-VARIABLE) (var-name part)
  (dolist (scope view)
    (and (scope-check-internal `(,scope) part)
         (return `(,var-name ,scope ,(nth (position scope view) value))))))

(defun set-variables-error (no variable view)
  (let ((error-reason  (case no
                         (1 (format nil "ERROR Variable ~A scope illegal." variable))
                         (2 (format nil "ERROR Variable ~A illegal." variable))
                         (3 (format nil "ERROR Variable ~A Undefined." variable)))))
    `(,variable ,view ,error-reason)))
```

```
(defun change-command-view (view)
  (case view
    (global      :global)
    (interactive :interactive)
    (functional  :functional)
    (decomposing :decomposing)
    (superposing :superposing)))

;; exec Fired-rule command make parameter-variables-list

(defun make-parameter-variables-list (rule)
  (setq *p-variables* nil)
  (make-parameter-variables-list-internal rule))

(defun make-parameter-variables-list-internal (rule)
  (loop for x in rule
        when (and x (listp x))
          do (make-parameter-variables-list-internal x)
        when (and (eql (variable-type x) :parameter)
                  (not (member x *p-variables* :key #'car))
;                 (not (eql (get-parameter-value x) :unbound))
                  )
          do (setq *p-variables* `(,@*p-variables* (,x ,(get-parameter-value x))))
        finally (return *p-variables*)))

;;;
;;; make send command

(defun send-variables-value-list (variables-list)
  (setq *erase-code* "")
  (let ((s-variables-list (make-erase-cr-code variables-list)))
    (push-output-que `(variables-value ,*self-process-id* ,s-variables-list))))

(defun send-constraints (part)
  (setq *erase-code* "")
  (let ((s-constraints (make-erase-cr-code (get-constraints  part ))))
    (push-output-que `(constraints ,*self-process-id* ,part ,s-constraints))))
;    (send-command-to-display `(constraints ,*self-process-id* ,part ,s-constraints))))

(defun send-fired-rule (rule)
  (setq *erase-code* "")
  (let* ((s-rule (make-erase-cr-code rule))
         (para-list (make-parameter-variables-list rule))
         (s-list (when para-list
                   (setq *erase-code* "")
                   (make-erase-cr-code para-list))))
    (if s-list
        (push-output-que `(fired-rule ,*self-process-id* ,s-rule ,s-list))
        (push-output-que `(fired-rule ,*self-process-id* ,s-rule)))))


(defun send-current-exec-part (part)
  (push-output-que `(current-exec-part ,*self-process-id* ,part)))

(defun send-result-set-variables-value (variables-list)
  (setq *erase-code* "")
  (let ((s-variables-list (make-erase-cr-code variables-list)))
    (push-output-que `(result-set-variables-value ,*self-process-id* ,s-variables-list))))
;    (send-command-to-display `(result-set-variables-value ,*self-process-id* ,s-variables-li
st))))
```

```
;;; utility

;; get

(defmethod (get-adress-valiables-list VARIABLES) ()
  adress-variables)

(defmethod (get-status-variables-list VARIABLES) ()
  status-variables)

(defun get-constraints (part)
  (case part
    (decomposing *decomposing-part*)
    (functional  *functional-part*)
    (superposing *superposing-part*)))

;; make send code

(defun make-erase-cr-code (list)
  (let ((return-string ""))
    (loop for x in list
          for *erase-code* = ""
          do
      (erase-cr-code x)
      (setq return-string (format nil "~A~A" return-string *erase-code*)))
    (format nil "(~A)" return-string)))


;; Some predicate

(defun command-receive-ok (id)
  (and (eql *server-part* :practice)
       (eql *self-process-id* id)
       (eql *service-status* :enable)))

(defun command-receive-ok-2 (id)
  (and (command-receive-ok id)
       *suspend-flag*))

(defun debug-mode-p ()
  (eql *execution-mode* 'debug))
```

```
;;; -*- Mode: LISP; Base: 10; Syntax: Common-Lisp; Package: CSL -*-

;;;
;;; Some utility functions for Network Control.
;;;

;; for server utiliy

(defun server-push-command (host command)
  (let ((stream (get-stream-from-host host)))
    (if stream
        (push-output-buffer (cons stream command)))))

(defun push-command-output-buffer (command-list id)
  (let ((stream (get-stream-from-id id)))
    (if stream
        (push-output-buffer (cons stream command-list)))))

;; send command to display

(defun send-command-to-display (command)
  (server-push-command *displayer-name* command))


; for client utility

(defun response-check (response)
  (selector response string-equal
    ("SUCCESS"      T)
    ("ERROR"       nil)
    (otherwise     nil) ))

;; utility

(defun get-command-from-output-buffer ()
  (without-interrupts (pop *output-buffer*) ))

(defun get-command-from-input-buffer ()
  (without-interrupts (pop *input-buffer*) ))

(defun get-instance (key val)
  (loop for info in *csl-system-configuration-table*
        when (get-instance-internal info key val)
          do (return info)
        finally (return nil)))

(defmethod (get-instance-internal NW-MANAGER)(key val)
  (case key
    (:id   (when (eql val process-id) self))
    (:host (when (eql val load-host)  self)) ))

(defun get-stream-from-host (host-name)
  (let ((instance (get-instance :host host-name)))
    (when instance (get-info instance :stream))))

(defun get-stream-from-id (id)
  (let ((instance (get-instance :id id)))
    (when instance (get-info instance :stream))))

(defun get-host-from-id (id)
  (let ((instance (get-instance :id id)))
    (when instance (get-info instance :host))))

(defmethod (get-info NW-MANAGER)(key)
  (case key
    (:id     process-id)
    (:host   load-host)
    (:spec   target-spec)
    (:stream access-stream)
    (:status service-status) ))

;; check

(defmethod (status-check NW-MANAGER)(key)
  (case key
    (:server-host T)
    (:load-spec   (eql service-status :ready))
    (:init-system (or (eql service-status :ready)
```

```
                    (eql service-status :enable)))
    (:event       (eql service-status :enable))
    (:end         T) ))


;; change utility

(defun change-command-to-symbol (command)
  (loop for x in command
        for y = (read-from-string x)
        when (and y (listp y))
          collect (change-command-list-to-symbol y) into return-list
        when (symbolp y)
          collect (intern x 'csl) into return-list
        when (numberp y)
          collect y into return-list
        when (stringp y)
          collect y into return-list
        finally (return return-list) ))

(defun change-command-list-to-symbol (command-list)
  (loop for x in command-list
        when (and x (listp x))
          collect (change-command-list-to-symbol x) into return-list
        when (symbolp x)
          collect (intern (string x) 'csl) into return-list
        when (numberp x)
          collect x into return-list
        when (stringp x)
          collect x into return-list
        finally (return return-list)))

(defun change-command-to-string (command)
  (loop for var in command
        when (stringp var)
          collect (format nil "~s" var) into return-list
        when (and var (listp var))
          collect (change-command-to-string var) into return-list
        unless (or (stringp var) (and var (listp var)))
          collect var into return-list
        finally (return return-list)))


; initialize

(defun network-init ()
  (kill-process-internal '(display send-client csl-protocol))
  (setq *display-process-id* nil
        *client-of-display*  nil
        *input-buffer*  nil
        *self-process-id*  nil                          ; ??
        *csl-system-configuration-table* nil ))

(defun client-init ()
  (setq  *output-buffer* nil
         *server-part*   nil ))

(defun server-init ()
  (setq *service-status*  nil
        *file-list*       nil                           ; spec file name
        *read-file-list*  nil
        *self-process-id* nil ))

(defun reset-debugger-flag ()
    (setq *execution-mode* 'normal
          *suspend-flag*    nil
          *step-exec-in*    T
          *step-exec-pro*   T ))


(defmethod (table-info-change NW-MANAGER)(key val)
  (case key
    (:service-status (setq service-status val))
    (:stream         (setq access-stream val)) ))


;; Kill Process
```

```
(defun kill-process (&optional (flg nil))
  (kill-process-internal '(in out pro))
  (and *timer-list*  (kill-timer-list))
  (unless flg (kill-process-internal '(send-client)))
  (setq *in-process*  nil
        *out-process* nil
        *pro*         nil ))

(defun kill-process-internal (list)
  (let ((process-list si:all-processes)
        (comfirm nil))
;    (if (member 'csl-protocol list)
;       (kill-process-csl-protocol))
    (dolist (process process-list)
      (if (member (send process :name) list :test #'string=)
          (si:com-kill-process process :confirm comfirm)))))

(defun kill-process-csl-protocol ()
  (let ((name 'csl-protocol)
        (process-list si:all-processes)
        (comfirm nil))
    (dolist (process process-list)
      (if (string= name (send process :name) :end2 (string-length name))
          (si:com-kill-process process :confirm comfirm)))))


(defun practice-ready-p ()
  (loop when (practice-ready-internal)
        do (return T)))

(defun practice-ready-internal ()
  (loop for x in *csl-system-configuration-table*
        unless (eql (get-info x :status) :enable)
          do (sleep 1) (return nil)
             finally (return t)))
```

```
;;; -*- Syntax: Common-Lisp; Base: 10; Package: CSL -*-

;;;
;;; DEBUGGER for hyouji-kei
;;;

;;
;; defvar
;;
(defvar *constraints-stream*     nil)
(defvar *fired-rule-stream*      nil)
(defvar *variables-value-stream* nil)
(defvar *change-part-stream*     nil)
(defvar *etc-stream*             nil)

(defvar *fired-rule-part*        nil)

(defun fired-rule-internal (id rule parameter)
  (format *fired-rule-stream* "~%    P-id ~a  FIRED RULE => ~a " id rule)
  (push (format nil "~%    P-id ~a  FIRED RULE => ~a " id rule) *fired-rule-part*)
  (when parameter
    (format *fired-rule-stream* " parameter-variables =>  ")
    (loop for var in parameter
          do
      (format *fired-rule-stream* " ~a " var)))
  (format *fired-rule-stream* "~%"))


(defun current-exec-part-internal (id part)            ;
    (format *change-part-stream* "~%    P-id ~a  After Change part => ~a~%" id part)
    (push (format nil "~%    P-id ~a  After Change part => ~a~%" id part) *fired-rule-part*))

(defun variables-value-internal (id variables-list)
  (display-variables-value-list id variables-list))

(defun result-set-variables-value-internal (id variables-list)
  (display-variables-value-list id variables-list))

(defun constraints-internal (id part constraints)
    (format *constraints-stream* "~%    P-id ~a  Part ~a  Rules ~%" id part)
    (loop for rule  in constraints
          do
      (format *constraints-stream* "        ~a ~%" rule)))

(defun display-variables-value-list (id variables-list)
  (format *variables-value-stream* "~%    P-id ~a  Variables ~%" id)
  (loop for variables in variables-list
        do
    (if (stringp (third variables))
        (format *variables-value-stream* "SET VARIABLES ~a <= (~a ~a) ~%" (third variables)
                (first variables) (second variables))
        (format *variables-value-stream* "        ~s ~%" variables))))


(defun set-output-mode (&optional flg)
  (unless flg
      (setq *constraints-stream*     t
            *fired-rule-stream*      t
            *variables-value-stream* t
            *change-part-stream*     t
            *etc-stream*             t)))

(set-output-mode)
```

```
;;; -*- Package: CSL; Syntax: Common-Lisp; Base: 10 -*-

(defvar *send-out-buffer* nil)

(DW:DEFINE-PROGRAM-FRAMEWORK CSL-DEBUGGER
  :SELECT-KEY
  #\h
  :COMMAND-DEFINER
  T
  :COMMAND-TABLE
  (:INHERIT-FROM '("colon full command" "standard arguments" "input editor compatibility")
   :KBD-ACCELERATOR-P 'NIL)
  :STATE-VARIABLES
  NIL
  :PANES
  ((PANE-1 :TITLE :HEIGHT-IN-LINES 1 :REDISPLAY-AFTER-COMMANDS NIL)
   (PANE-2 :DISPLAY                              ;constraints
           :more-p nil
           :typeout-window t
           :margin-components
           '((dw::margin-borders :thickness 1)
             (dw::margin-scroll-bar :margin :left )
             (dw:margin-label :string "Constraints"
                              :style (:fix :roman :large)
                              :margin :top)))
   (PANE-3 :DISPLAY                              ;fired-rule
           :more-p nil
           :margin-components
           '((dw::margin-borders :thickness 1)
             (dw::margin-scroll-bar :margin :left )
             (dw:margin-label :string "Fired rule"
                              :style (:fix :roman :large)
                              :margin :top)))
   (PANE-4 :DISPLAY                              ;change-part
           :more-p nil
           :margin-components
           '((dw::margin-borders :thickness 1)
             (dw::margin-scroll-bar :margin :left )
             (dw:margin-label :string "Change part"
                              :style (:fix :roman :large)
                              :margin :top)))
   (PANE-6 :DISPLAY                              ;variables-value
           :more-p nil
           :margin-components
           '((dw::margin-borders :thickness 1)
             (dw::margin-scroll-bar :margin :left )
             (dw:margin-label :string "Variables value"
                              :style (:fix :roman :large)
                              :margin :top)))
   (PANE-5 :DISPLAY                              ;etc.
           :more-p nil
           :margin-components
           '((dw::margin-borders :thickness 1)
             (dw::margin-scroll-bar :margin :left )
             (dw:margin-label :string "ETC."
                              :style (:fix :roman :large)
                              :margin :top)))
   (PANE-7 :INTERACTOR :HEIGHT-IN-LINES 4))
  :CONFIGURATIONS
  '((DW::MAIN
     (:LAYOUT (DW::MAIN :COLUMN PANE-1 PANE-2 PANE-3 ROW-1 PANE-5 PANE-7)
      (ROW-1 :ROW PANE-4 PANE-6))
     (:SIZES
       (DW::MAIN (PANE-1 1 :LINES) (PANE-7 4 :LINES) :THEN (PANE-2 :EVEN) (PANE-3 :EVEN)
                 (ROW-1 :EVEN) (PANE-5 :EVEN))
       (ROW-1 (PANE-4 :EVEN) (PANE-6 :EVEN))))))

;;;

;; global variables
;; for stream defvar in :>sakane>atr>csl>network>debugger-hyoujikei.lisp

(defvar *ready* nil)
(defvar *read-error* nil)


;; define command
```

```
(define-csl-debugger-command (com-clear) ()
   (clear-internal))

(define-csl-debugger-command (com-stream-init) ()
   (stream-init))

(define-csl-debugger-command (com-start)
  ((file-name 'string :prompt "Configuration table : "))
   (csl-init)
   (test-top2 file-name))

(defun test-top2 (file-name)
  (network-init)
  (client-init)
  (load-configuration-file file-name)
  (setq *server-part* :display
        *display-process-id* (process-run-function  'display #'display-start2)
        *client-of-display* (process-run-function 'send-client #'send-client-command) )
  (send-all-practice :server-host)
  (when (or (practice-ready-p) *read-error*)
    (unless *read-error*
      (setq *ready* t)
      (my-draw-string "All practice ready.~%" *etc-stream*))))

(define-csl-debugger-command (com-init-system) ()
   (if *ready*
       (init-system-internal)
       (my-draw-string "Not Ready.~%" *etc-stream* )))

(defun init-system-internal ()
  (let ((to-id (accept 'symbol :prompt "Enter id ")))
    (send-one-practice :init-system to-id)))

(define-csl-debugger-command (com-event) ()
   (if *ready*
       (event-internal)
       (my-draw-string "Not Ready.~%" *etc-stream*)))

(defun event-internal ()
  (let ((command (accept 'list :prompt "Enter event ")))
   (send-one-practice :event (third command) command)))

(define-csl-debugger-command (com-append-load-spec) ()
  (if *ready*
      (let ((id (accept 'symbol :prompt "Which id "))
            (spec (accept 'list :prompt "Enter spec-file-list ")))
        (send-append-load-spec id spec))
      (my-draw-string "Not Ready.~%" *etc-stream*)))

(define-csl-debugger-command (com-show-constraints)
   ((id   'symbol :prompt "Which id ")
    (part  '((member decomposing functional superposing)) :prompt "part" :default nil))
   (send-show-constraints id part))

(define-csl-debugger-command (com-show-variables-value)
   ((id   'symbol :prompt "Which id "))
   (send-show-variables-value id))

(define-csl-debugger-command (com-set-mode)
   ((id   'symbol :prompt "Which id ")
    (mode  '((member normal debug)) :prompt "mode" :default 'normal))
   (send-set-mode id mode))

(define-csl-debugger-command (com-end) ()
   (send-all-practice :end)
   (sleep 3)
   (test-end)
   (setq *ready* nil))

(define-csl-debugger-command (com-suspend-event-transaction)
   ((id   'symbol :prompt "Which id "))
   (send-suspend-event-transaction id))

(define-csl-debugger-command (com-resume-event-transaction)
   ((id   'symbol :prompt "Which id "))
   (send-resume-event-transaction id))

(define-csl-debugger-command (com-show-exec-part)
```

```
   ((id   'symbol :prompt "Which id "))
   (send-show-exec-part id))

(define-csl-debugger-command (com-invoke-next-event-transaction)
   ((id   'symbol :prompt "Which id "))
   (send-invoke-next-event-transaction id))

(define-csl-debugger-command (com-set-variables-value) ()
   (if *ready*
       (let ((id (accept 'symbol :prompt "Which id "))
             (var-list (accept 'list :prompt "Enter var-view-value-list ")))
         (send-set-variables-value id var-list))))


(defun display-start2 ()
  (setq *send-out-buffer* nil
        *fired-rule-part* nil)
  (loop when *input-buffer*
          do
      (let* ((command (get-command-from-input-buffer))
             (to-id (third command))
             (from-id (fifth command))
             (com-string (format nil "    OUT==>~S~%" command)))
        (my-draw-string com-string *etc-stream*)
        (push com-string *send-out-buffer*)
        (unless (or (eql to-id from-id) (eql to-id 'device) (not (get-instance :id to-id))
                    (eql from-id 'device))
          (push-output-buffer
            (cons (get-info (get-instance :id to-id) :stream) command) )
          (my-draw-string (format nil "    SEND OUT==>~S~%" command) *etc-stream*)))))

;;; utility

(defun csl-init ()
  (setq *ready*         nil
        *read-error*    nil)
  (or (frame-mode-p) (stream-init))
  (clear-internal))

(defun my-draw-string (string stream)
;  (or (frame-mode-p)(stream-init))
  (format stream string))

(defun clear-internal ()
;  (or (frame-mode-p) (stream-init))
  (send *constraints-stream* :clear-history)
  (send *fired-rule-stream* :clear-history)
  (send *variables-value-stream* :clear-history)
  (send *change-part-stream* :clear-history)
  (send *etc-stream* :clear-history))

(defun frame-mode-p ()
  (equal *constraints-stream* (dw:get-program-pane 'pane-2)))

(defun stream-init ()
  (setq *constraints-stream*     (dw:get-program-pane 'pane-2)
        *fired-rule-stream*      (dw:get-program-pane 'pane-3)
        *variables-value-stream* (dw:get-program-pane 'pane-6)
        *change-part-stream*     (dw:get-program-pane 'pane-4)
        *etc-stream*             (dw:get-program-pane 'pane-5)
        *send-out-buffer* nil
        *fired-rule-part* nil))

; rel-7-2>flavor>examine-window.lisp

;; patch

(defun read-error (command)
  (let* ((process-id (car (change-command-to-symbol (cdr command))))
         (instance (get-instance :id process-id)))
    (setq *read-error* t)
    (my-draw-string (format nil "Process-id   : ~s~%" process-id) *etc-stream*)
    (my-draw-string (format nil "File-name    : ~s~%" (third command)) *etc-stream*)
    (my-draw-string (format nil "Error reason : ~s~%" (fourth command)) *etc-stream*)
    (table-info-change instance :service-status :ready)))
```