マルチメディア通信と分散処理 58−13
（1992. 11. 19）

MSCによるLOTOS仕様の記述方法

安藤津芳† 太田正孝† 高橋薫‡

†高度通信システム研究所 ‡東北大学

本論文では、これまで通信ソフトウェア設計の補助情報として使われてきたメッセージシーケンスチャートからLOTOSで記述された仕様に解釈する方法とその評価について述べる．まず本論文で使用するメッセージシーケンスチャートの定義を示した後に，LOTOS仕様への解釈方法とその評価を述べる。解釈は、単一のメッセージシーケンスチャートをLOTOS記述に変換した後に統合するという方法を取っており、現状では個々の変換より統合に多く問題が残っているが、試用した結果の総合的な評価では、これまでの様に直接エディタ等を用いて仕様記述言語で記述するよりも、遥かに効率的で漏れのないものができることがわかった。

Description methods of a LOTOS specification using MSC

Tsuyoshi Ando† Masataka Ohta† Kaoru Takahashi‡

†AIC System Laboratories

6-6-3,MinamiYoshinari,Aoba-ku,Sendai-shi, Miyagi,989-32 JAPAN

‡Tohoku University

2-1-1,Katahira,Aoba-ku,Sendai-shi,Miyagi,980 JAPAN

ABSTRACT This paper proposes translation methods from Message Sequence Charts to Formal Description Language, LOTOS. The first section shows translation objects and their background. The second section defines our Message Sequence Chart which is subset of CCITT Message Sequence Chart. The third section proposes translation methods from Message Sequence Charts to LOTOS. The last section shows the estimation of the methods and future studies.

1. はじめに

近年，ソフトウェア開発に仕様を形式的仕様記述言語を用いて定義することで自動プログラミングや自動検証といった機械支援を期待すると共に，仕様の曖昧性の除去･欠落の防止を行う要求が強くなってきている．これに対応して，通信ソフトウェアの世界でもCCITT, ISOによって形式的仕様記述言語SDL[1]やLOTOS[2], Estelle[3]が勧告されており，さらにはその機能拡張が検討されている．一方，通信ソフトウェアの開発では，これまで自然言語による仕様定義の補助的なものとして，状態遷移図･状態遷移表･シーケンスチャート等が使われてきた．中でもシーケンスチャートは，MSC (Message Sequence Chart)として，通信ソフトウェアの仕様記述に，広く用いられようとしている．[4],[5]

そこで，本論文では，これまで仕様検証能力が高いという点で着目されていたにもかかわらず，仕様記述者からその記述性･理解性といった面から敬遠されがちであったLOTOS仕様を，これまで仕様定義の補助的役割で使用されていたシーケンスチャートに詳細化を補助する階層化記述機能を加えて，シーケンスチャートによる仕様からLOTOS仕様のひな型を生成する方法を提案する．この目的は，単にLOTOS仕様記述への誘いにとどまらず，LOTOSの持つ高い検証能力[2],[6],[7],[8]を利用して，仕様の等価性または上位互換性を保った詳細化を補助することである．本提案を用いた場合の仕様記述の流れを図1に示す．

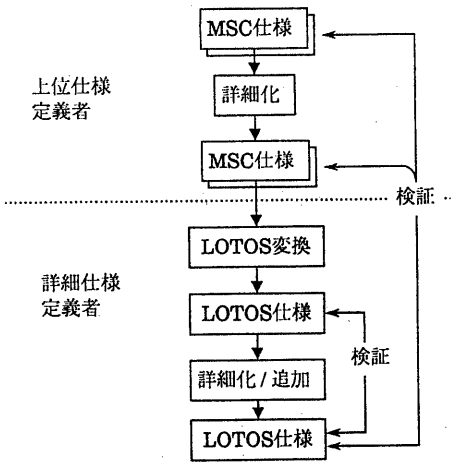


図. 1 本論文の提案に基づく仕様記述の流れ

以下，本論文では，まず対象とするMSCを定義し，次にLOTOS解釈の方法を提案し，評価する．

2. MSCの定義

本論文で取り扱うMSCは，一方向単一通信を基本としたMSCとする．本MSCの意味付けは次のものである．

各通信は，二者間に限定され，一回の通信では一つのmessageのやり取りに限定されるMSCである．この場合，各通信では，送信者と受信者が確定される．なお，各通信は同期通信とする．

また，MSCによる仕様の詳細化を可能にするために階層化記述機能を追加する．

これらの条件を満たすように，以下にテキスト表現でのMSCの定義を拡張BNFを用いて示す．

[MSC(テキスト表現)の定義]

```
<MSC Definition>      ::= <message sequence chart>
                          {<submsc>}*
<end>                 ::= [ <note> ] < semicolon>
                          [ <note> ]
<note>                ::= /* <character string> */
< semicolon >         ::= ;
<message
  sequence chart>     ::= msc <msc head><msc body>
                          endmsc <end>
< msc head >          ::= <message sequence chart name>
                          <end>
                          [ < msc interface > ]
<msc interface >      ::= {inst <instance list > |
                          env <environment list > } *
                          <end>
<instance list >      ::= <instance name>
                          [, <instance list > ]
<environment list >   ::= <environment name>
                          [,<environment list>]
  (注) ここでの<environment name>とは，MSCの最上位
       シーケンス(msc内にある)インスタンス名がenvまた
       はenv_<name>のものとする．
< msc body >          ::= < msc section >*
< msc section >       ::= [<global condition>]
                          { <instance definition> }*
                          [<global condition>]
<instance definition> ::= instance <instance head>
                          <instance body> endinstance
                          <end>
<instance head>       ::= { [ <qualifier> ] <instance
                          name> <end>
                          [ decomposed <end> ]}|
                          {[<qualifier>]
                          <environment name><end>}
<qualifier>           ::= < path item>
                          [/ <path item >]*
< path item>          ::= <name>
<instance body>       ::= < instance event list>
                          [ <stop> ]
< instance event list>::= { <message input > |
                          < message output > |
                          <condition > }*
< message input >     ::= in <msg identification>
                          from <address> <end>
< message output >    ::= out <msg identification>
                          to <address> <end>
<msg identification>  ::= <message name>
                          [, <message instance name> ]
<address>             ::= <instance name> |
                          <environment name>
< global condition >  ::= condition <condition name >
                          <end>
< condition >         ::= condition <condition name >
                          [ shared { <shared instance list>
                          |all}] <end>
<shared instance list>  ::=  <instance name >
                          [, <shared instance list> ]
< stop >              ::= stop
```

< submsc >　　::= submsc[<qualifier>]
　　<msc head ><msc body >
　　endmsc <end>

□

なお，ここで用いた定義は，その通信方式が同期通信か非同期通信かということを除けば，CCITTで勧告化されるMSC[4]のサブセットである．また，ここで定義したテキスト表現を図形表現に対応させるのは，インスタンスの階層化記述を除いてはCCITTの勧告案と同一の方式で可能である．図形表現におけるインスタンスの階層化は，詳細化時に常に上位のシーケンスを参照しながら編集できる様に変更したものである．図形表現の具体例を図2に示す．

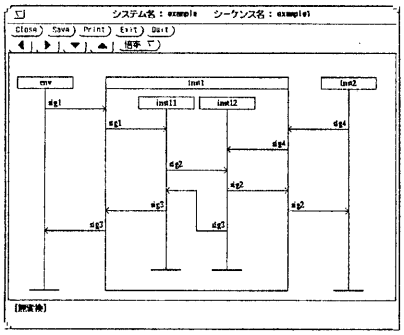


図2. MSCの図形表現例

次に，階層化を取り入れたことによるインスタンスとメッセージのスコープを次のように定める．

- インスタンスのスコープ
  各インスタンスから参照可能なものは，先祖・兄弟・子供とし，それ以外の部分で同一名称のものが有ってもかまわない．
- メッセージのスコープ
  メッセージ名に関してはすべてグローバルなものとする．

また，シーケンスにおける時間の流れは，インスタンス毎に異なり，通信時にのみ相手のインスタンスと同期しているものとする．

3. LOTOS解釈方法

本章では，前章で定義したMSCをLOTOS仕様にするための解釈の方法を述べる．
解釈時の主な課題として次のものが挙げられる．

① インスタンス個別の動作解釈
② 階層構造と通信の解釈
③ 複数のMSCから一つの仕様への統合

これらの課題において，①と②に関しては，一つのMSCをどのようなLOTOS仕様に解釈するかの課題であり，③は，解釈した結果の，複数のLOTOS仕様の統合法である．

そこで，これらの課題を，それぞれどのように解決するかを含めて以下に示す．ここでの基本的な項目として，MSCのLOTOS解釈と，複数のMSCからの一つの仕様への統合という二つの処理が有る．これらを行う順番として，まず各MSCを，それぞれ個別にLOTOS解釈し，その後に一つの仕様に統合する方法を取る．この理由は，MSC個別解釈のほうが簡単であり，その解釈結果の正当性確認が簡単であることと，MSC上で統合してもMSCの本来持っている長所である理解性を損なうだけでメリットがないためである．もちろんどちらを先に行ったとしても，統合の正当性確認という課題は残っている．

本節では，この解釈の順番に従い，その方法を先に示した課題の解決を含めて説明する．

3.1 単独MSCの解釈

ここで述べる単独MSCの解釈として次の二種類の方法を提案する．

- 最詳細下位インスタンス優先の解釈
  上位のシーケンスは設計者の便宜上のものであると考え，この部分は仕様の階層化情報としてのみ生かす．一方，プログラムとして実現されるのは，一般には最詳細下位インスタンスの処理であることから，最詳細下位インスタンスの処理を中心に解釈する．
- 上位シーケンスによる制約的解釈
  上位のシーケンスは，仕様の階層化を定義しているだけでなく，そのレベルの信号シーケンスに制約を与えるものであると解釈し，最詳細下位インスタンスの処理だけでなく，上位で与えられた制約を反映させて解釈する．

この二種類の解釈を考慮した理由は，前者が，構造化設計・プログラミング過程で，設計者とプログラマが異なった場合に行われがちな解釈であり，後者は，設計の詳細化における情報の欠落を防止したものであるからである．

以下，これら二つの解釈方法について説明する．

3.1.1 最詳細下位インスタンス優先の解釈

本項では，単独のMSCの最詳細下位インスタンス優先の解釈方法について，先に示した解釈上の課題①，②をからめて述べる．

解釈の手順として，表1に示すステップに分けて行うこととする．

表1. 単独のMSCの解釈手順

ステップ	実　行　内　容	対応課題
1	構造の解釈	②
2	type定義の生成	②
3	Gateの決定と同期条件の抽出	②
4	インスタンス個別の動作解釈	①

次に，各ステップに対応させて解釈方法を示す．

(1) 構造の解釈

環境(env)インスタンスを除いて，すべてのインスタンスを，LOTOSのプロセスと解釈して，MSC仕様におけるLOTOSのプロセス構造を算出する．具体的には，図2に示したMSCからは図3のような階層が算出できる．

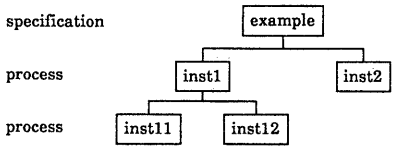


図3. 構造の算出例

(2) type定義の生成

typeとして必要な定義は，メッセージに対するものである．今回対象としたMSCにおけるメッセージのスコープはグローバルと制限しているので，図4で示すひな型に対してmessage-set部分をMSCからメッセージ名を抽出して展開する．

```
type message_id is
   sorts message_id
   opns message-set : → message_id
endtype
```

図4. message type定義のためのひな型

(3) Gateの決定と同期条件の抽出

Gateの決定と同期条件の抽出を検討する上で，重要なポイントとなるものとして通信制御のやり方がある．即ち，集中方式を取るか分散方式を取るかということである．これに関しては，集中制御のほうがインプリメントは簡単では有るが，解釈結果に制御のためのプロセスが追加されるためにユーザにとって理解しにくくなる．そこで，分散方式を取ることにする．

次に，同期関係を機械処理する上でLOTOSの並列処理の重要な特徴を示す．

[特徴]

あるプロセスが，並列に実行する複数のプロセスの生成を行う場合に，それらのプロセスの持つ同一名のゲートはすべて同期するという条件の下では，以下の帰納的な記述が可能である．

(i) 生成するプロセスインスタンスが1つの場合または，最初のプロセスインスタンスの場合

$Bex_1[gate\_list_1] = inst_1[gates_1]$

ここで，$gate\_list_1 = gates_1$ とする．

(ii) 二つめ以後のプロセスインスタンスを追加する場合

$Bex_i[gate\_list_i] = inst_i[gates_i] \ |[sync\_gates_{i-1}]| \ Bex_{i-1}[gate\_list_{i-1}]$

ここで，$gate\_list_i = gates_i \cup gate\_list_{i-1}$

また，$sync\_gates_{i-1} = gates_i \cap gate\_list_{i-1}$

とする．($sync\_gates_{i-1} = \phi$ の時は$|||$を用いる)

□

この特徴を利用すれば，プロセス構造を反映させた同期ゲートの算出とそれに基づく通信制御は，ゲート名の付与を考慮するだけでよい．

そこで，ゲート名の付与方法を考える．最も簡単な方法は，通信毎にユニークなゲート名を付与することであるが，これではゲート数が膨大になりまとまりがつかなくなる危険を含んでいるだけでなく，統合を考慮した場合には，効率的な処理ができない．そこで，今対象としているMSCは，二者間の通信しか含んでいないので，最終的に通信を行う最詳細階位インスタンス二者間毎にゲートを一つ付与してやれば，効率的に実現でき，統合時にも使用できる．

この規則に基づいて，先の図2のMSCを解釈すると次のような結果を得ることができる．

```
specification example[env_inst11]:noexit
  type message_id is
    sorts message_id
    opns sig1,sig2,sig3,sig4 : → message_id
  endtype
  behaviour
    hide inst12_inst2 in
      inst1[env_inst11,inst12_inst2]
      |[inst12_inst2]|
      inst2[inst12_inst2]
    where
      process inst1[env_inst11,inst12_inst2]:noexit :=
        hide inst11_inst12 in
          inst11[env_inst11,inst11_inst12]
          |[inst11_inst12]|
          inst12[inst11_inst12,inst12_inst2]
        where
          process inst11[env_inst11,inst11_inst12]
                                              :noexit :=
            ...
          endproc
          process inst12[inst11_inst12,inst12_inst2]
                                              :noexit :=
            ...
          endproc
      endproc
      process inst2[inst12_inst2] :noexit :=
        ...
      endproc
endspec
```

(4) インスタンス個別の動作解釈

ここで対象とするのは，それぞれのMSCにおける最詳細階位インスタンスの処理である．定義したMSCにおいて最詳細階位インスタンスに現れる動作の表現は次のものである．

- メッセージの送受信
- 状態名
- インスタンスの終結

それぞれについて解釈の方法を示す．

i) メッセージの送受信

メッセージの送受信については，それぞれのメッセージについて，その送受信に関与するゲートをMSCから算出し，送受信種別に対応して次のように変換する．

[送信時]

<ゲート名>!<信号名>

[受信時]

<ゲート名>?<変数名>:message_id

ここで，送信側が環境(env)の場合に限って,入力を保証をするためにガードをかける．

<ゲート名>?<変数名>:message_id

[<変数名> = <信号名>]

ii) 状態名

インスタンス内に状態が現れた場合には，各状態

をそのインスタンスの内部インスタンスとして定義し，インスタンシエーションによる状態遷移を実現する．

iii)インスタンスの終結

各インスタンスの最後の処理が**condition**の場合には，その状態のインスタンシエーションとする．インスタンスの終結(stop)が現れた場合やそれ以外の場合には，一律LOTOSの**stop**に変換する．

以上は基本的な変換であるが，これらを実現した後，LOTOSシンタックスとの整合を取るために，内部処理iや**action prefix**";"を必要に応じて補う．また，各プロセスのファンクショナリティについても便宜上，noexitに統一して変換する．

これらの変換規則を図2の例に適用した結果を次に示す．

```
specification example[env_inst11]:noexit
  type message_id is
    sorts message_id
    opns sig1,sig2,sig3,sig4 : → message_id
  endtype
  behaviour
    hide inst12_inst2 in
      inst1[env_inst11,inst12_inst2]
      |[inst12_inst2]|
      inst2[inst12_inst2]
    where
      process inst1[env_inst11,inst12_inst2]:noexit :=
        hide inst11_inst12 in
          inst11[env_inst11,inst11_inst12]
          |[inst11_inst12]|
          inst12[inst11_inst12,inst12_inst2]
        where
          process inst11[env_inst11,inst11_inst12]
                                              :noexit :=
            env_inst11?x:message_id[x = sig1];
            inst11_inst12!sig2;
            inst11_inst12?y:message_id;
            env_inst11!sig3; stop
          endproc
          process inst12[inst11_inst12,inst12_inst2]
                                              :noexit :=
            inst12_inst2?x:message_id;
            inst11_inst12?y:message_id;
            inst12_inst2!sig2;inst11_inst12!sig3;stop
          endproc
      endproc
      process inst2[inst12_inst2] :noexit :=
        inst12_inst2!sig4;inst12_inst2?x:message_id;stop
      endproc
endspec
```

3.1.2 上位シーケンスによる制約的解釈

本項では，もう一つの単独MSCの解釈の方法である上位シーケンスによる制約的解釈について説明する．ただし，この解釈は，その手順を始めとする主な部分は前項の最詳細階位インスタンス優先の解釈と同じであるので，ここでは差分中心に説明する．

まず，前項の解釈との差分を示す．差分は，ゲート名の付与方法と上位シーケンスのインスタンスの解釈の部分だけである．それぞれの解釈方法について以下に述べる．

(1) ゲート名の付与方法

ゲート名は，それぞれのレベルのシーケンスにおいて通信を行っている二者のインスタンスに対応して付与する．

また，詳細化が行われているインスタンスに対しては，内部インスタンスとの通信を実現するために外部と通信を行う内部インスタンス毎に自分との通信のための内部ゲートを付与する．この時，内部インスタンス毎の通信は，同一レベルのシーケンスとなるので，先の規則によりゲートが内部ゲートとして付与される．

(2) 上位シーケンスのインスタンスの処理の解釈

まず，上位シーケンスにおけるインスタンスの処理の解釈を，前項の最詳細階位インスタンスの処理の解釈と同様の方法で行う．

次に，外部のインスタンスからメッセージを受信した場合には，その受信を行う内部インスタンスへ送信する様に，受信処理の直後に加える．

さらに，外部にインスタンスへメッセージを送信する場合には，その送信を行っている内部インスタンスからの受信を行うように，送信処理の直前に追加する．

以上を行って完成させたインスタンスの処理を，内部インスタンスと通信ができるように，インスタンシエーション部に前項で示した特徴を利用して追加する．

以上が，前項の解釈との差分である．具体的に先の図2のMSCに対して 本解釈を適用した場合の例を次に示す．

```
specification example[env_inst1]:noexit
  type message_id is
    sorts message_id
    opns sig1,sig2,sig3,sig4 : → message_id
  endtype
  behaviour
    hide inst1_inst2 in
      inst1[env_inst1,inst1_inst2]
      |[inst1_inst2]|
      inst2[inst1_inst2]
    where
      process inst1[env_inst1,inst1_inst2]:noexit :=
        hide self_inst11,self_inst12,inst11_inst12 in
          inst11[self_inst11,inst11_inst12]
          |[self_inst11,inst11_inst12]|
          inst12[self_inst12,inst11_inst12]
          |[self_inst12,self_inst12]|
          ( env_inst1?x:message_id[x=sig1];
            self_inst11!x;
            inst1_inst2?y:message_id[y=sig4];
            self_inst12!y;self_inst12?z:message_id;
            inst1_inst2!z;self_inst11?z1:message_id;
            env_inst1!z1; stop )
        where
          process inst11[g1,g2]:noexit:=
            g1?x:message_id;g2!sig2;
            g2?y:message_id;g1!sig3;stop
          endproc
          process inst12[g1,g2]:noexit:=
            g1?x:message_id;g2?y:message_id;
            g1!sig2;g2!sig3;stop
          endproc
      endproc
```

```
    process inst2[inst1_inst2] :noexit :=
      inst1_inst2!sig4;inst1_inst2?x:message_id;
      stop
    endproc
endspec
```

3.2 解釈後LOTOS仕様の統合

仕様の統合は，構造情報から対応するレベルを決定し，そのレベルで和統合することを基本とする．ここで，和統合とは，構造情報から対応するレベルを決定し，そのレベル単位で仕様間の記述内容をOR論理にしたがって統合することである．

各レベルでの方針を簡単に示す．

[specificationレベル]

外部とのgate，ファンクショナリティの決定(noexit優先)，hideするgate，初期起動プロセス群とその間の同期関係の整理を行う．

[上位processレベル]

上位とのgate，ファンクショナリティの決定，hideするgate，初期処理，起動プロセス群とその間の同期関係の整理を行う．

[最詳細階位processレベル]

- 起動元情報からネスト構造の整合を行う．
- 上位processとgateの整合を行う．
- 処理を比較し，異なっている場合にはchoiceによる分岐を行う．この時，分岐条件は決定できないので，コメント付加によりユーザに注意を促す．

最後に内部処理iやaction prefix';'や優先度制御のための括弧'('')'を挿入する．

統合の手順は，前節の二種類の単独MSCの解釈の方法のどちらを使用しても同一の方法で可能である．そこで，ここでは最詳細階位のインスタンス優先の解釈を例に説明する．

具体例として，図2のMSCから最詳細階位のインスタンス優先の解釈を用いて作成したLOTOS仕様と，図5に示すMSCから同一の解釈によって作成したLOTOS仕様の統合結果を以下に示す．

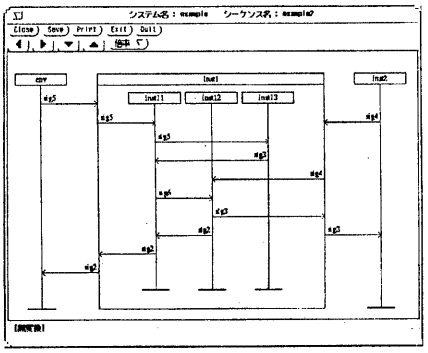


図5. MSCの例(その2)

```
specification example[env_inst11]:noexit
  type message_id is
    sorts message_id
    opns sig1,sig2,sig3,sig4,sig5,sig6 : →message_id
  endtype
  behaviour
    hide inst12_inst2 in
      inst1[env_inst11,inst12_inst2]
      |[inst12_inst2]|
      inst2[inst12_inst2]
    where
      process inst1[env_inst11,inst12_inst2]:noexit :=
        hide inst11_inst12,inst11_inst13 in
          inst11[env_inst11,inst11_inst12,inst11_inst13]
          |[inst11_inst12,inst11_inst13]|
          inst12[inst11_inst12,inst12_inst2]
          |||
          inst3[inst11_inst13]
        where
          process inst11[g1,g2,g3]:noexit :=
            ( g1?x:message_id[x=sig1];
              g2!sig2;g2?y:message_id[y=sig3];
              g1!sig3; stop )
            []
            ( g1?x:message_id[x=sig5];g3!sig5;
              g3?y:message_id[y=sig3];g2!sig6;
              g2?z:message_id[z=sig2];g1!sig2;stop )
          endproc
          process inst12[g1,g2]:noexit :=
            g2?x:message_id[x=sig4];
            ( ( g1?y:message_id[y=sig2];
                g2!sig2;g1!sig3;stop)
              []
              ( g1?y:message_id[y=sig6];
                g2!sig3;g1!sig2;stop ))
          endproc
          process inst3[g1]:noexit:=
            g1?x:message_id[x=sig5];g1!sig3;stop
          endproc
      endproc
      process inst2[g1] :noexit :=
        g1!sig4;
        ( ( g1?x:message_id[x=sig2];stop )
          []
          ( g1?x:message_id[x=sig3];stop ))
      endproc
endspec
```

ここでは，前述の単純MSCの解釈に次の二点変更を加えている．

- 各シーケンスを意識できるようにインスタンス間の通信の受信に対してもガードを付加．
- 最詳細階位インスタンスのゲートは，スペースの関係から名称を最適化．

ここで，後者はともかく，前者の変更は，統合矛盾を避ける上で必要である．

4. 解釈の評価

本章では，これまで説明したMSCの解釈を

- 単独MSCの解釈の正当性
- 統合の正当性
- ユーザにとっての使いやすさ
- 本解釈の制約
- 解釈後仕様

を尺度として，評価を行う．

4.1 単独MSCの解釈の正当性

単独MSCの解釈の正当性については，本来は証明が必要であるが，一般的な証明は困難である．そこで，ここでは，いくつかの解釈後のLOTOS仕様をシミュレーションした結果を用いて評価する．

まず，最詳細下位インスタンス優先の解釈の場合については，図2を変換した例をシミュレータ[9]で実行すると次の結果を得ることができる．

```
0    0    START
1    1        env_inst11 !sig1
2    2            i(inst12_inst2) !sig4
3    3            i(inst11_inst12) !sig2
4    4            i(inst12_inst2) !sig2
5    5            i(inst11_inst12) !sig3
6    6        env_inst11 !sig3
```

これは，一見，図2のMSCを満たす様に思えるが，次のような問題を含んでいる．この解釈では，構造的な階層は持ってはいるが処理に関しては最詳細階位インスタンスしか意識していないので，処理上は図6のような変換を行っている．従って，上位の定義で

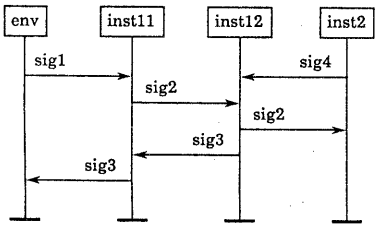


図6. 図2の最詳細階位インスタンスのシーケンス

信号の順序を定義している(図7参照)にもかかわらず，処理が並列に実行可能となり，本来シーケンスにないものを含んでいる(sig1とsig4の順番が非決定的である)．

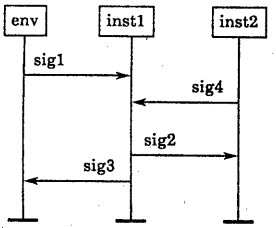


図7. 図2の上位シーケンス

ここで示した問題は，解釈の問題であると共に，一般の設計の詳細化でも起こしやすい問題である．そこで，この解釈はこのような問題点の洗い出しに使用できると思われる．

一方，上位シーケンスによる制約的解釈での実行結果としては次のものを得ることができる．

```
0    0    START
1    1        env__inst1 !sig1
2    2            i(self__inst11) !sig1
3    3            i(inst1__inst2) !sig4
4    4            i(self__inst12) !sig4
5    5            i(inst11__inst12) !sig2
6    6            i(self__inst12) !sig2
7    7            i(inst11__inst12) !sig3
8    8            i(inst1__inst2) !sig2
9    9            i(self__inst11) !sig3
10   10       env__inst1 !sig3; stop
```

これを見ると，6と7の実行順序が異なるが，この部分は元の図2のシーケンスであっても並列的な部分であり，実行順は規定されていない．そしてこの部分以外には並列性を含んではいない．即ち，図2のシーケンスに忠実に対応した仕様といえる．もちろん，ユーザが並列処理を要求する場合には，別のシーケンスを記述して統合するようにすればよい．

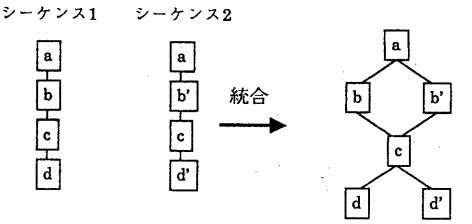
4.2 統合の正当性

統合の正当性の評価についても，単独MSCの解釈と同様に，例から変換されたLOTOS仕様のシミュレーション結果として比較し，評価する．

統合は，単独MSCの解釈の方法にとらわれないので，最詳細階位インスタンス優先の解釈を用いて行う．前述の統合例をシミュレーションすると次の結果を得ることができる．

```
0    0    START
1    1        env_inst11 !sig1
2    2            i(inst12_inst2) !sig4
3    3            i(inst11_inst12) !sig2
4    4            i(inst12_inst2) !sig2
5    5            i(inst11_inst12) !sig3
6    6        env_inst11 !sig3
7    1        env_inst11 !sig5
8    2            i(inst12_inst2) !sig4
9    3            i(inst11_inst13) !sig5
10   4            i(inst11_inst13) !sig3
11   5            i(inst11_inst12) !sig6
12   6            i(inst12_inst2) !sig3
13   7            i(inst11_inst12) !sig2
14   8        env_inst11 !sig2
```

この場合には，入力信号によって処理が別れているので，矛盾が発生していないが，同一入力で出力が異なる場合等は，MSCの持つ情報だけでは処理の決定ができない．つまり，図8に示すような現象が発生



本来の
a-b-c-d, a-b'-c-d' だけでなく，
a-b'-c-d, a-b-c-d'
も，実行可能にしている．

図8. 統合による問題点

し，本統合により，実際には実行されない処理シーケンスを生成するという問題を持っている．

従って，上の問題を如何に解決するかがより有効な解釈にするための鍵となる．現在の解釈では，こ

のような場合には非決定的分岐にコメントによるシーケンス種別を付けて対処しているが，さらなる検討が必要である．

4.3 ユーザにとっての使いやすさ

今回定義したMSCは，一般的に使用されているものとの差はあまり無い．このため，LOTOSを知らない設計者であってもMSCを知っていれば，仕様検証を可能とするLOTOS仕様を作成できるという点で有効である．また，MSCを知らなくてもそれを理解することは，LOTOSを理解することよりはるかに簡単である．

従って，図1に示した様に，MSCとLOTOSの使用者が別れる場合にはLOTOSでの検証を前提として，設計者に有効な補助を可能としている．

4.4 本解釈の制約

本解釈の持ついくつかの制約の中で，問題となると考えられるのは，次のものである．

① ゲート名を自動付与しているので，名称指定や分割といった自由度がない
② メッセージにパラメータを許していない
③ メッセージをグローバルに扱っているので，階層化ができない．
④ LOTOSの持つ多重同期を生かした仕様ができない．

まず，①については，現状ではLOTOS変換後の修正となる．これに関してはLOTOSを知っている人にとっては不便であるかもしれないが，LOTOSを意識しなくてもLOTOS仕様ができるという目的に反するのでしかたがないと割り切ったが，この変更はMSCのエディタのレベルで可能である．次に，②，③についてはメッセージtypeの持ち方で，解決できる見通しが立っている．④については，MSCを多重同期を表示できるように改良すれば，解釈はほぼ同じ方法で可能であるが，MSCの簡明さを殺すものになってしまうため，その必要性を含め再考する．

以上の点を除けば，今回の解釈はある程度実用に耐えるものである．

4.5 解釈後仕様

ここでは，解釈後のLOTOS仕様の記述スタイル[10]，規模，理解性について評価する．

まず，LOTOS記述スタイルに関しては，本解釈が行っているのは，MSC上でのインスタンスの意味付にもよるが，基本的にはリソース指向による構造分割の後に状態指向を用いた記述である．これは，LOTOS記述において一般に良いとされている制約指向とは異なる．しかし，その記述スタイルの特徴から，最終的に実装を意識した仕様を作成するという目的に対応したのもである．

次に，解釈後の仕様規模を見てみると，最詳細階位インスタンス優先の解釈はもちろんのこと上位シーケンスによる制約的解釈であっても，階層を意識したLOTOS記述と比較した場合には，その記述量は2倍を越えないと予想される．この差分は，自動解釈であることを考えれば，十分満足のできる範囲である．

更に，解釈後のLOTOS仕様の理解性を見てみると，最詳細階位インスタンス優先の解釈では，並列に動作するプロセス間の通信先が明確であり，通信を行うプロセス間の関係が把握しやすいが，階層別の処理の理解は困難である．一方，上位シーケンスによる制約的解釈では，各通信ゲートが階層に閉じているので，最詳細階位のインスタンス間の直接的な通信は理解しにくいが，階層別の処理が明白である．従って，段階的な階層化において，上位仕様や下位仕様との比較が簡単であり，解釈後の設計者によるLOTOS仕様の具体化・処理追加の助けになる．

5. まとめ

本論文では，上位仕様定義者と詳細仕様定義者の間の情報のやり取りを円滑にすることで高信頼な通信ソフトウエア開発をすることを目的として，MSCを用いてLOTOS仕様を作成するための解釈の方法と，その評価を行った．評価の結果，提案した解釈は，まだいくつかの問題を含んではいるが，当初の目的を支援するのに十分なものであることが判った．

今後の課題としては，評価の項目で述べたいくつかの問題点の解決と，本方式に基づいた支援系の構築がある．

謝辞

最後に，本研究を行うにあたり，数々の有益な助言をして頂いた東北大学野口正一教授・白鳥則郎教授に深謝致します．また，本研究の機会を与えてくださった当社の緒方常務，ならびに日頃有益な検討をして頂く当研究所の研究員の皆様に感謝致します．

[参考文献]

[1] CCITT : "Functional Specification and Description Language (SDL)", Recommendation Z. 100 (1989)

[2] ISO : "Information processing system - Open systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour", ISO 8807 (1989)

[3] ISO : "Information processing system - Open systems Interconnection - Estelle - A formal description technique based on an extend state transition model", ISO 9074 (1989)

[4] CCITT : " Message Sequence Chart (MSC)" , Recommendation Z. 120 (1992)

[5] CCITT : "Functional Specification and Description Language (SDL) - Methodorogy Guidance - ", Recommendation Z. 100 Appendix 1 (1992)

[6] E. Brinksma, G. Scollo and C. Steenbergen : " Lotos Specifications, their Implementations and their Tests", Proc. IFIP WG6. 1, 6th Int. Workshop on Protocol Specification, Testing and Verification , North-Holland, pp.349-360 (1987)

[7] H. Ichikawa, K. Yamanaka and J. Kato : " Incremental specification in LOTOS", Proc. IFIP WG6. 1, 10th Int. Workshop on Protocol Specification, Testing and Verification, North-Holland, pp.185-200 (1990)

[8] K. Yamano, D. Jokanovic, T. Ando, M. Ohta and K. Takahashi : " Formal Specification and Verification of ISDN Services in LOTOS", IEICE Transactions on Communication, E75-B, 8 (1992)

[9] P. H. J. van Eijk, C. A. Vissers and M. Diaz : " The Formal Description Technique LOTOS ", North-Holland (1989)

[10] C. A. Vissers, G. Scollo and M. V. Sinderen : "Architecture and Specification Style Formal Description of Distributed Systems", Proc. IFIP WG6. 1, 8th Int. Workshop on Protocol Specification, Testing and Verification, North-Holland, pp.189-204 (1988)