プログラミングー言語・基礎・実践ー 5－7
(1991. 11. 21)

LOTOSによるソフトウエアプロセスの全体記述と開発者個人ごとのプロセス記述の導出

中山 高史　　東野 輝夫　　谷口 健一

大阪大学 基礎工学部 情報工学科

あらまし 一般にソフトウエアプロセスの全体記述では，開発に参加している技術者各人の仕事の内容と，各仕事の全体としての実行順序だけが記述されている．その場合，この記述から，技術者各人が，どのようなタイミングで，誰とどのようなデータやメッセージを交換しながら，各人の仕事を進行すればよいかの記述（“個人のプロセス記述”と呼ぶ）を，自動的に導出できれば望ましい．本報告では，Marc Kellnerによって提案されたソフトウエアプロセスの共通問題（ISPW6の例題）の全体記述を分散システムの仕様記述言語LOTOSで記述し，この記述から，開発に参加する技術者各人のプロセス記述を自動的に導出し，導出の際の問題点や得られた個人のプロセス記述の有効性等について調べている．導出アルゴリズムの概略についても述べる．

Derivation of Software Process Description for each Developer from Whole Software Process Description written in LOTOS

Takashi NAKAYAMA, Teruo HIGASHINO and Kenichi TANIGUCHI
Dept. of Information and Computer Sciences,
Faculty of Engineering Science, Osaka University
Toyonaka, Osaka, 560 JAPAN

Abstract In a software process description, the contents of the works of developers in a group and an order of executing them are written. But, because of complication, it is not described (1)when each developer must do his works, and (2)when and whom he must send the data which other developers use, and so on. It is desirable that a personal process description of each developer containing the latter description can be derived from the former one. In this paper, we wrote a whole description of Kellner's Software Process Example in LOTOS, and derived personal process descriptions for all developers from the whole description. The outline of the derivation algorithm is also explained.

1 まえがき

ソフトウエアの開発，保守，変更などの作業行程といったものを広義の計算プロセスとして考え，これらのプロセスを形式的に記述するといった，ソフトウエアプロセスに関する研究が最近関心を集めている [1 - 4]．ソフトウエアプロセスを形式的に記述することの目的には，ソフトウエアの開発行程における技術者各人の作業内容の明確化や，開発時間の算出などがあるが，従来提案されていた形式的な記述方法では，開発に参加している技術者各人のすべき仕事と，それらの実行順序だけが記述されており，開発工程において，"作業の終了を別の人に伝える"，あるいは"作成したデータを必要な人に渡す"等の技術者間で必要な連絡のやり取りまでは，記述が複雑になるために記述されていない．そのため開発行程において技術者各人が，どのようなタイミングで，誰とデータやメッセージを送受信しながら，各人の仕事(イベント)を進行すればいいのかを，記述全体から読みとるのは難しい．そこで技術者各人ごとに，自分の仕事内容，技術者間での仕事の同期の取り方，技術者間の連絡作業などを含んだ作業手順("個人のプロセス記述"と呼ぶ)を，ソフトウエアプロセスの全体記述から自動的に導出すれば，開発行程における個人の作業がより明確となり，開発を円滑に行なうことができる．

開発工程において，技術者各人を分散システムと考え，技術者間のデータのやり取り等をシステム間の通信作業と考えれば，ソフトウエアプロセスの記述は，分散システムの記述に類似している．LOTOS は分散システムの形式記述言語として，1989 年に ISO(国際標準化機構) によって規格化され [5]，近年 LOTOS を用いて多くの分散システムが記述されている [11]．また LOTOS を用いてソフトウエアプロセスの記述を行なう研究も進められている [6][10]．さらに LOTOS のサブクラスで書かれた全体記述から，各個人のプロセス記述を自動的に導出する方法についての研究も進められている [7][13][14] [15]．若干の制約はあるが，我々は文献 [7] において，Full-LOTOS で書かれた全体記述から個人のプロセス記述を導出するアルゴリズムを提案している．本研究では，ソフトウエアプロセスの共通問題として取り上げられている Kellner の例題仕様 (ISPW6 の例題)[8] を LOTOS で形式的に記述し，その記述から，文献 [7] のアルゴリズムを使って，開発に携わる技術者個人のプロセス記述を自動的に導出した．そして導出の際の問題点や，得られた個人プロセスの有効性等について調べた．

以下，2 章では LOTOS について，3 章では Kellner の例題の説明及び LOTOS での全体記述について，4 章では個人のプロセス記述について，5 章では個人のプロセス記述の導出方法の概略，6 章では導出アルゴリズムについて，それぞれ説明する．

2 LOTOS

LOTOS は並列実行や選択実行，割り込みなどが記述できる分散システムや通信プロトコルの形式記述言語で，動作単位をイベントと呼び，イベントの生起順序の記述を動作表現と呼ぶ．各動作表現は，次の (1)～(9) のイベント，オペレータ，プロセスを組み合わせて定義する．LOTOS の仕様は 1 つの主プロセスと複数のサブプロセスの組からなる．LOTOS におけるデータ型の記述は代数的言語 ACT ONE を用いて行われる．

1. イベント　$a?x : typex?....!E1!, ...[Q]$
("a" はゲート名，"?" は入力，"!" は出力，"Q" はこのイベントが実行可能であるための条件 (イベントの実行条件))

2. **終了イベント　exit**

3. **連接**　$a; B$
(イベント a の実行後，動作表現 B のイベントが実行可能)

4. **選択実行**　$B1[]B2$
(動作表現 B1,B2 のいずれかが実行可能)

5. **非同期並列実行**　$B1|||B2$
(動作表現 B1,B2 は非同期並列実行可能)

6. **同期並列実行**　$B1|[g1, ..., gn]|B2$
(動作表現 B1,B2 は並列実行可能．但し B1,B2 中のイベント g1,...,gn は同時に実行されなければならない)

7. **順次合成**　$B1 >> B2$
(動作表現 B1 の全イベントの実行後 B2 のイベントが実行可能)

8. **割り込み**　$B1[> B2$
(B1 のイベント実行中 B2 の最初のイベントが実行されれば，以降 B2 のイベントのみ実行可能)

9. **プロセス**　$P[a, b, ...](x, y, ...) := B$
(P が呼び出されると動作表現 B のイベントが実行される．"a,b,..." は B に現れるイベント名．"x,y,..." はプロセスパラメタ (P を呼び出す際の形式パラメタ)．B にプロセス名が含まれてもよい)

3 Kellner の例題仕様

kellner の例題仕様 (ISPW6 の例題) は，Marc Kellner[8] の提案した，ソフトウエアプロセスモデリングにおける様々なアプローチの理解や比較を行なうための例題であり，実世界のソフトウエアプロセスにおける種々の問題を含んでいるため，多くの異なったカテゴリに属する問題をモデル化することができる．例題は一つの核問題 (core problem) と幾つかの拡張問題 (optional extension) から構成されている．核問題はソフトウエアプロセスの様々なモデルにおける共通した問題であり，拡張問題は多様な方法で問題を解くことができるよう核問題を拡張したものである．本研究ではこの核問題に焦点を当て，形式的記述を行なう．核問題ではソフトウエア開発中にソフトウエアの要求仕様が変更されたと仮定した上で，それに対してどのような変更作業を行なうかが設定されている．変更プロセスは 8 個の部分プロセスに分かれており，各々の部分プロセスにおいて，実行すべき内容が説明されている．各プロセスの説明は，プロセスで行なうべき仕事の総括的なものと，プロセスでの入出力，プロセス実行にあたっての責任者や制限が定義されている．ただし各プロセスの内容はあくまで総括的なものであり，具体的な内容までは説明されていない．また各プロセスの実行順序関係も一部明確に指定されていない部分がある．このため曖昧な部分は，実際のソフトウエア変更作業で一般に考えられる作業順序に従って定義を行なった．

3.1 例題の設定

Kellner の例題仕様では，ソフトウエア開発段階において，何らか問題で要求仕様を変更した場合において，変更した要求仕様 (以下では要求変更と呼ぶ) に従ってソフトウエアを変更，修正するといった状況を考えている．変更作業に参加する技術者と役割を次に示す．

- Project Manager(PM)
プロジェクト全体を管理，進行する責任者
- Software Engineer(SE)
DE，PG の仕事を管理，検査するエンジニア
- Design Engineer(DE)
設計とコーディングに責任を持つ設計エンジニア

- Quality Assurance Engineer(QAE)
テストの作成と実行に責任を持つ品質保証エンジニア
- Programmer(PG)
プログラムを作成するエンジニア

実際の開発においてはSE, DE, QAE, PGの役割は複数の人で構成されているのが普通であるが，ここでは問題を簡略にするために各1名づつに設定する．変更作業は，要求変更をPMが受理して始まる．PMは要求変更から変更作業の役割分担と作業スケジュールを作成する．変更工程における個々の作業は，8個の部分作業(サブプロセス)に分かれており，変更したソフトウエアが最終的なテストに合格することにより，作業が終了する．

3.2 サブプロセスの定義

変更には様々な作業があるが，変更作業の工程を次の8個のサブプロセスに分けて定義する．以下では各プロセスの作業内容，プロセスの実行者，プロセスの入力と出力について述べる．プロセスの説明の最後に，そのプロセスをLOTOSで記述した例を示す．本研究の記述では，ゲート名に技術者各人の名前を割り当てている．ゲートの種類としては，技術者の名前であるゲート(PM, SE, DE, QAE, PG)の他に，技術者間でのデータやメッセージの送受信用のゲート(PMG, SEG, DEG, QAEG)，予め用意された各種のデータを入力するためのコンピュータ用のゲート(CP)，プロセス間での同期をとるためのゲート(Wait)がある．以下にゲートの関係を図で示す．

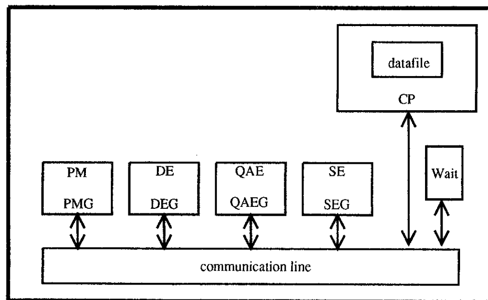


図1. ゲートの関係図

各サブプロセスの実行順序は後述のメインプロセスで指定されている．また各サブプロセスの入力としては，プロセスにおける変数引数と，コンピュータ用のゲートを通してコンピュータに入力される変数の2種類がある．また出力は各々のサブプロセスの実行者の，送受信用のゲートを通して行なう．

1.Schedule and Assign Tasks(計画の作成と作業割り当て)

このプロセスでは，ソフトウエア変更に関する作業の計画とスケジュールを作成する．作成はPMが行ない，このプロセスは，すべてのプロセスの最初に実行される．

入力 1. 要求変更(req) 2. 変更前の作業計画(old_project_plan)

出力 1. 修正した作業計画(project_plan) 2. スケジュール日程(task_date)

make_planは，req(要求変更)とold_project_plan(変更前の作業計画)から，新しい作業計画を作成する関数で，make_dateは，req(要求変更)とold_project_plan(変更前の作業計画)から，スケジュール日程を作成する関数である．これらの関数の実現法については，抽象レベルでは記述しない．以下関数の説明は省略する．

```
process Schedule_and_Assign_Tasks
[PM,PMG,DE,QAE,PG,SE](req:file,old_project_plan:file):=
(PM?project_plan:file
[project_plan=make_plan
(req,old_project_plan)];exit
||| PM?task_date:file
    [task_date=make_date
    (req,old_project_plan)];exit)
>>(DE!project_plan!task_date;exit  |||
   PG!project_plan!task_date;exit  |||
   QAE!project_plan!task_date;exit |||
   SE!project_plan!task_date;exit)
>> PMG!project_plan !task_date;exit
endproc
```

図2.Schdule and Assign Tasksの記述例

2.Modify Design(デザインの修正)

このプロセスでは，要求変更により影響を受ける単体コードユニットのデザインの修正を行なう．修正されたデザインはReview Designで見直され，最後に修正されたデザインが生成される．またReview Designでもう一度デザインの修正が必要と判断された場合には，Review Designからのフィードバックに基づいたデザイン修正も行なう．このプロセスは主にDEによって実行される．

入力 1. 修正前のデザイン(old_design) 2.Review Designからのデザインのフィードバック(feed_back)(feed_backはReview Designからフィードバックがある場合はフィードバックされた内容，ない場合はnilである) 3. 要求変更(req)

出力 1. 修正したデザイン(modified_design)

Modify Designの記述例は，次のReview Designと関係があるため，Review Designで示す．

3.Review Design(デザインの見直し)

このプロセスでは，Modify Designで修正されたデザインの正式なレビュー(見直し)を行なう．レビューの結果デザインの修正が必要な場合には，レビューの結果をModify Designに送り，もう一度デザインの修正を行なう．このステップはSEによって実行される．

入力 1. 修正されたデザイン(design)

出力 1. レビューの結果(design_review_result) 2. デザインのフィードバック(design_review_feedback)

```
process Modify_Design
[DE,SE,DEG,SEG]
(old_design:file,feed_back:fback,req:file):=
DE?modified_design:file
[modified_design=modify_design
(old_design,feed_back,req)];exit
>> (Design_Review[SE,SEG](modified_design)|[SEG]|
    SEG?result:judge ?back:fback;exit)
>>((DEG!result[result=OK];DEG!modified_design;exit)
[] (DEG!result[result=NO];Modify_Design
    [DE,SE,DEG,SEG](modified_design,back)))
where
process Design_Review[SE,SEG](design:file):=
SE?design_review_result:judge
```

```
[design_review_result=review(design)];exit
||| SE?design_review_feedback:fback
   [design_review_feedback=make_feedback
   (design)];exit)
>> SEG!design_review_result
   !design_review_feedback;exit
endproc
endproc
```

図 3.Modify Design と Design Review の記述例

4.Modify Code(コードの修正)

このプロセスでは，修正されたデザインと既存のコードを基にコードを修正する．次に修正したコードをオブジェクトコードにコンパイルする．このステップは，最終的な単体試験(Test Unit)において，コードが不適切であると判断された場合にも行なわれる．このプロセスはPGによって実行される．

入力 1. 修正されたデザイン (design) 2. 修正前のコード (old_code) 3.Test Unit からのコードのフィードバック (feed_back)(feed_back は Test Unit からフィードバックがある場合はフィードバックされた内容，ない場合は nil である)

出力 1. 修正されたコード (modified_code) 2. オブジェクトコード (object_code)

```
process Modify_Code
[PG,PGG](design:file,old_code:file,feed_back:fback):=
PG?modified_code:file
[modified_code=modify_code
(design,feed_back,old_code)];
PG?object_code:file[object_code=make_object
(modified_code)];
PGG!modified_code!object_code;exit
endproc
```

図 4.Modify Code の記述例

5.Modify Test Plan(試験計画の修正)

このプロセスでは，作成されたデザインやコードなどに関する試験計画の修正を行なう．実際の試験データや手続きなどの修正は，Modify Unit Test Package で行なう．このプロセスはQAEによって実行される．

入力 1.修正前の試験計画 (old_test_plan) 2. 要求変更 (req)

出力 1.修正された試験計画 (modified_test_plan)

```
process Modify_Test_Plan
[QAE,QAEG](old_test_plan:file,req:file):=
QAE?modified_test_plan:file
[modified_test_plan=modify_plan(old_test_plan,req)];
QAEG!modified_test_plan;exit
endproc
```

図 5.Modify Test Plan の記述例

6.Modify Unit Test Package(単体試験パッケージの修正)

このプロセスでは，作成されたデザインやコードなどに関する様々な試験を修正する．試験する対象は色々あり，個々の試験は単体試験と呼ぶ．ここではすべての単体試験を一つのパッケージとして考える．このプロセスは，最終的な単体試験(Test Unit)において，単体試験パッケージが不適切であると判断された場合にも行なわれる．このプロセスはQAEによって実行される．

入力 1. 修正された試験計画 (引数 plan) 2. 修正されたデザイン (design) 3. 修正前の単体試験パッケージ (old_unit_test_package) 4.Test Unit からの単体試験パッケージのフィードバック (feed_back)(feed_back は Test Unit からフィードバックがある場合はフィードバックされた内容，ない場合は nil である)

出力 1. 修正された単体試験パッケージ (modified_unit_test_package)

```
process Modify_Unit_Test_Package
[QAE,QAEG](plan:file,design:file,
old_unit_test_package:file,feed_back:fback):=
QAE?modified_unit_test_package:file
[modified_unit_test_package=modify_package
(plan,design,old_unit_test_package,feed_back)];
QAEG!modified_unit_test_package;exit
endproc
```

図 6.Modify Unit Test Package の記述例

7.Test Unit(単体試験)

このプロセスでは，修正されたコード上での単体試験パッケージの実行と，その結果の解析を行なう．試験パッケージの実行がまず行なわれ，それにより試験結果が算出される．試験の結果を解析し，問題がないと判断された場合には，修正の再実行の必要なしとの結果を出力する．問題があると判断された場合には，原因としてソースコードの修正，単体試験パッケージの修正のどちらか一方が必要であるという結果をソースコードのフィードバック，あるいは単体試験パッケージのフィードバックと共に出力する．このプロセスはSEによって実行される．

入力 1. オブジェクトコード (code) 2. 修正された単体試験パッケージ (test_package)

出力 1. 試験結果 (test_result)("コードの修正が必要"，"単体試験パッケージの修正が必要"，"修正の必要なし" の3種類) 2. コードまたは単体試験パッケージのフィードバック (feed_back)(引数 feed_back は，コード修正が必要な場合はコードのフィードバックの内容，単体試験パッケージの修正が必要な場合は単体試験パッケージのフィードバックの内容，修正の必要がない場合は nil である)

```
process Test_Unit[SE,SEG]
(code:file,test_package:file):=
SE?test_result:string
[test_result=make_result(code,test_package)];
SE?feed_back:fback
[feed_back=make_fback(code,test_package)];exit
>> (SEG!test_result!feed_back; exit)
endproc
```

図 7.Test Unit の記述例

8.Monitor Progress(進行状況の監視)

このプロセスでは修正作業中において，何らかの理由で修正作業をもう一度最初からやり直すか，修正作業を中止する．このプロセスはPMによって実行される．
入力，出力 なし

```
process monitor_Progress
[PM,DE,QAE,PG,CP,PMG,DEG,QAEG,PGG,SEG,Wait]:=
(PM?Retry:oral; ISPW6
[PM,DE,QAE,PG,CP,PMG,DEG,QAEG,PGG,SEG,Wait])
[]PM?Stop:oral;exit
endproc
```

図 8.Monitor Progress の記述例

3.3 サブプロセスの実行順序

前述の 8 個のサブプロセスの実行順序とデータのやり取りを次に示す．各プロセスは並列，または順次に実行される．

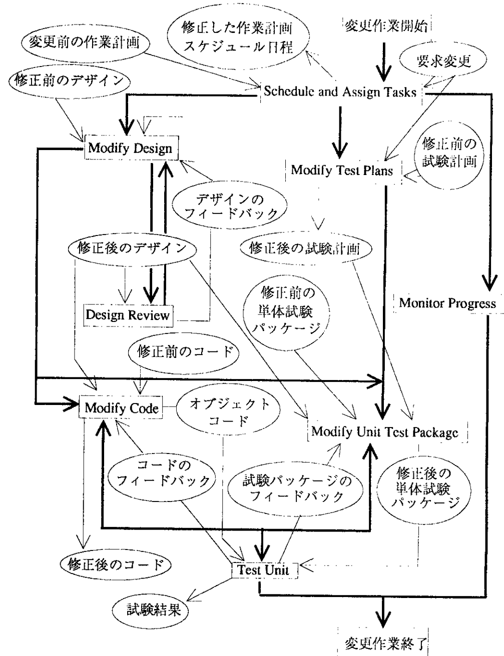


図 9. サブプロセスの実行関係

3.4 例題の全体記述

前述の 8 個のサブプロセスの実行関係に従って，例題仕様を LOTOS で記述した例を以下に示す．図 9 において，サブプロセス Test Unit と Modify Code, Modify Unit Test Package の間で作業の繰り返しがあるが，Test Unit でコードの修正が必要と判断された場合に，まず Modify_code をでコードの修正を行なった後に，Test_Unit を実行し，その後 3 つのサブプロセス間で繰り返し実行を行なうのが Remake_Code であり，単体試験パッケージの修正が必要と判断された場合に，まず Modify_Unit_Test_Package で単体試験パッケージを修正した後に，Test_Unit を実行し，その結果により 3 つのサブプロセス間で繰り返し実行を行なうのが Remake_Package である．

```
specification
ISPW6[PM,DE,QAE,PG,SE,CP,PMG,DEG,QAEG,PGG,SEG,Wait]
behaviour
ISPW6[PM,DE,QAE,PG,SE,CP,PMG,DEG,QAEG,PGG,SEG,Wait]:=
CP?requirement_change:file;
CP?original_project_plan:file;
CP?original_design:file; CP?original_code:file;
CP?original_test_plan:file;
CP?original_unit_test_package:file;exit
>> (Schedule_and_Assign_Tasks[PM,PMG,DE,QAE,PG,SE]
   (requirement_change,original_project_plan)|[PMG]|
     PMG?project_plan:file ?task_date:file;exit)
 >>(((((Modify_Design[DE,SE,DEG,SEG]
        (original_design,nil,requirement_change)|[DEG]|
         DEG!modified_design;Wait;exit)
     ||| Modify_Test_Plan[QAE,QAEG]
        (original_test_plan,requirement_change)
 |[QAEG]|QAEG?test_plan:file;exit)
 >> Modify_Unit_Test_Package[QAE,QAEG]
   (test_plan,modified_design,
    original_unit_test_package,nil)
 |[QAEG]|QAEG?modified_test_package:file;exit)
 |[Wait]| (Wait; Modify_Code[PG,PGG]
          (modified_design,original_code,nil)
 |[PGG]| PGG?modified_code:file ?object_code:file;exit))
 >>Test_Unit[SE,SEG](object_code,modified_test_package)
 |[SEG]|SEG?test_result:string ?test_feed_back:fback;exit)
 >>(([test_result=''needless''] -> SEG;exit)
   []([test_result=''code''] -> SEG;
       Remake_Code[PG,PGG,SE,SEG,QAE,QAEG]
      (modified_design,modified_test_package,modified_code,
       test_feed_back,object_code,test_plan))
   []([test_result=''package''] -> SEG;
      Remake_Package[QAE,QAEG,SE,SEG,PG,PGG]
      (test_plan,modified_design,
       test_feed_back,object_code,modified_test_package,
       modified_code)))
 >> PM?End;exit
 [> Monitor_progress
   [PM,DE,QAE,PG,SE,CP,PMG,DEG,QAEG,PGG,SEG,Wait]
 where
 process Schedule_and_Assign_Tasks:=...
 process Modify_Design:=...
 process Design_Review:=...
 process Modify_Code:=...
 process Modify_Test_Plan:=...
 process Modify_Unit_Test_Package:=...
 process Test_Unit:=...
 process Monitor_Progress:=...

 process Remake_Code[PG,PGG,SE,SEG,QAE,QAEG]
 (design:file,package:file,s_code:file,feed_back:fback,
  o_code:file,test_plan:file):=
 Modify_Code[PG,PGG](design,s_code,feed_back)|[PGG]|
 PGG?m_s_code:file ?m_o_code:file;
 Test_Unit[SE,SEG](m_o_code,package)|[SEG]|
 SEG?t_result:string ?re_feed_back:fback;exit
 >> (([t_result=''code'']-> SEG;
      Remake_Code[PG,PGG,SE,SEG,QAE,QAEG]
      (design,package,m_s_code,re_feed_back,
       m_o_code,test_plan))
   []([t_result=''package'']->
      Remake_Package[QAE,QAEG,SE,SEG,PG,PGG]
      (plan,design,re_feed_back,m_o_code,package,m_s_code))
   []([t_result=''needless'']->
      (SEG!t_result; exit)))
endproc
process Remake_Package[QAE,QAEG,SE,SEG,PG,PGG]
(plan:file,design:file,feed_back:fback,o_code:file,
 package:file,s_code:file):=
Modify_Unit_Test_Package[QAE,QAEG]
(plan,design,package,feed_back)|QAEG|QAEG?m_package:file;
Test_Unit[SE,SEG](o_code,m_package)|[SEG]|
SEG?t_result:string ?re_feed_back:fback;exit
>>(([t_result=''code'']-> SEG;
    Remake_Code[PG,PGG,SE,SEG,QAE,QAEG]
    (design,m_package,s_code,re_feed_back,o_code,plan))
 []([t_result=''package'']-> SEG;
    Remake_Package[QAE,QAEG,SE,SEG,PG,PGG]
    (plan,design,re_feed_back,o_code,m_package,s_code))
```

```
[]([t_result=''needless'']->
   (SEG!t_result;exit)))
endproc
endspec
```

図 10.Kellner の例題仕様の全体記述例

4 個人のプロセス記述

ここでは 3.2.3 の例題記述のサブプロセス Modify_Design を使って個人のプロセス記述とはどのような記述か，また全体記述に比べてどのように違うかについて説明する．

4.1 個人のプロセス記述に必要な情報

プロセス Modify_Design では，まず元のデザイン old_design とフィードバック情報 feed_back を基に，DE が修正したデザイン modified_design を作成する．次に SE はプロセス Design_Review を実行すると同時に result と back を作成する．result="OK" の場合には，DE が modified_design を出力する．また result="NO" の場合には，プロセス Modify_Design を再び実行する．プロセス Modify_Design からは以上の仕事を DE,SE が指定された順にそれぞれ実行するという情報のみが記述されているが，実際には，次の図 11 で示すように，同期用のメッセージや変数の値を，技術者間で送受信しなければならない．(図 11 では，ゲート DEG は DE の，ゲート SEG は SE の入出力用のゲートなので，DEG, SEG のイベントはそれぞれ DE,SE のイベントとして考える)

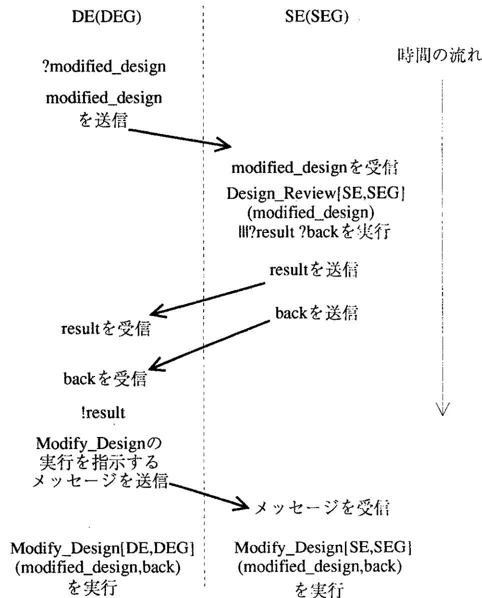


図 11. 技術者各人に必要な情報
(result=NO の場合のみ記述)

DE は modified_design を入力したら，それを次に使う SE に送らなければならない．DE は modified_design を受信した後に，プロセス Design_Review を実行する．また SE は result と back を入力した後に，変数とその値を，それらを次に使う人 (result，back とも DE) に送らなければならない．SE は，result の値が "OK" の場合には，次に作業を実行する DE に対して，同期用のメッセージを送らなければならない．また result の値が "NO" の場合には，次にプロセス Modify_Design を実行する DE に，そのことを伝えなければならない．プロセス Modify_Design の例は，記述が短いので，各自のすべき連絡作業が全体記述から比較的容易に読み取れるが，記述が複雑な全体記述からでは，他の技術者との連絡作業を正確に把握するのは難しい．したがって，全体記述から把握しづらい連絡作業やイベント実行の同期などを，個人のプロセス記述に書くことにより，より作業が円滑に行なえる．

4.2 個人のプロセス記述例

個人のソフトウエアプロセス記述について，3.2 のプロセス Modify_Design に対する個人のプロセス記述の例を用いて説明する．以下の例では，技術者 "p" に対して，"Mi" というメッセージを送信することを "S_p(Mi)"，変数 "x" とその値を送信することを "S_p(x)" で表し，"p" からメッセージ "Mi" を受信することを "R_p(q)"，変数 "x" とその値を受信することを "R_p(x)" で表す．メッセージ番号 "i" はメッセージの種類を表す番号である．以下メッセージを M1,M2，変数を x,y のように記述する．

- プロセス Modify_Design での DE の作業

  プロセス Modify_Design において DE はまず modified_design を入力する．次に modified_design を SE に送る．その後 SE から変数 result と back とそれらの値が送られてくるので，受信する．result が "OK" の場合には，result を出力してから，SE にプロセス終了を伝えるメッセージ (M1) を送った後に modified_design を出力する．result が "NO" の場合には，result を出力してから，SE に Modify_Design の実行を指示するメッセージ (M2) を送った後にプロセス Modify_Design[DE,DEG] を実行する．

```
process Modify_Design[DE,DEG]
(old_design:file,feed_back:fback):=
DE?modified_design:file[...]
;S_SE(modified_design);exit
>>(R_SE(result);exit|||R_SE(back);exit)
>>((DEG!result[result=OK];
    S_SE(M1);DE!modified_design;exit)
 [](DEG!result[result=NO];
    S_SE(M2);Modify_Design[DE,DEG]))
endproc
```

図 12.Modify_Design での DE 個人のプロセス記述

- プロセス Modify_Design での SE の作業

  プロセス Modify_Design において SE はまず DE から変数 modified_design とその値が送られてくるので，それを受信する．次にプロセス Design_Review の実行と，result，back を入力した後に result と back を DE に送信する作業を，同時に行なう．その後 result が "OK" であることを伝えるメッセージ (M1) か result が "NO" であることを伝えるメッセージ (M2) を DE から送られてくるので，M1 を受信した場合には，作業を終了する．M2 を受信した場合には，プロセス Modify_Design[SE,SEG] を実行する．

```
process Modify_Design[SE,SEG]
```

```
(old_design:file,feed_back:fback):=
R_DE(modified_design)
>> (Design_Review[SE,SEG](modified_design)
|[SEG]|(SEG?result:judge?back:fback;
(S_DE(result);exit|||S_DE(back);exit)))
>>(R_DE(M1);exit[]R_DE(M2);Modify_Design[SE,SEG])
endproc
```

図 13.Modify_Design での SE 個人のプロセス記述

5 個人のプロセス記述導出の概略

全体記述から個人のプロセス記述を導出する方法の概略を説明する．我々は，Full-LOTOS で書かれた全体記述から，個々のプロセスを自動的に生成するアルゴリズムを研究，開発した [7]．そこでこのアルゴリズムを用いて，全体記述から個々の技術者のプロセスを導出する．ここでは Full-LOTOS で記述された全体記述中に現れる動作表現を幾つかのタイプに分類し，各タイプごとに，どのように個人記述を導出するかを説明する．全体記述中のある特定の技術者 "p" の個人記述の導出は，基本的には，与えられた全体記述から "p" で実行されるイベントのみを取り出し，それらの前後に，イベントの実行順序を制御する同期用メッセージの送受信動作を加えるという方法で行う．また，LOTOS の各プロセスに現れるデータの値は，そのデータを最初に使うイベントを実行した際に決まるので，そのイベントの実行後，データの値を必要とする技術者に送るようにした．本研究では，個人のプロセス記述を導出する際に幾つかの制約条件があるが，それについては文献 [7] に詳しく述べられている．

5.1 $A(x);B(x)$(連接と変数) タイプの記述

"PM?x;SE?y;DE!x!y;exit" という動作表現を考える．(A(x)="PM?x", B(x)="SE?y;DE!x!y;exit" とし，A(x);B(x) がプロセス P に含まれているとする)．連接オペレータでは，A というイベントの実行を行なった後に，B の最初のイベントを実行する人に，作業終了を伝えるメッセージを送る．B のイベントを実行する人は，このメッセージを受信した後に，B のイベントを実行する．また PM?x のように A において，変数 x を最初に使うイベント (変数 x の値を最初に定めるイベントと呼ぶ) の実行者 (この場合 PM) は，プロセス P の中で変数 x を使うすべての人に，変数 x とその値を送るようにする．プロセス P が P:=A(x);B(x) の場合，PM は PM?x の実行後 B の最初のイベント実行者である SE に PM?x の終了を伝える同期用メッセージを送り，B の中で x の値を必要とする DE に変数 x とその値を送る．このプロセスにおいて，PM, SE, DE それぞれの作業を記述すると次のようになる．

PM:PM?x;((S_SE(M1);exit)|||(S_DE(x);exit))
SE:R_PM(M1);SE?y;S_DE(y);exit
DE:R_PM(x);R_SE(y);DE!x!y;exit

5.2 $A[]B$(選択実行) タイプの記述

"PM?x;SE?y;DE?z;exit[]PM?v;DE?w;exit" に対して，オペレータ ";" の前後だけに作業終了を通知する同期用メッセージの送受信動作を挿入しただけの個人記述を示すと次のようになる．

PM:PM?x;S_SE(M1);exit[]PM?v;S3(M2);exit
SE :R_PM(M1);SE?y;S_DE(M3);exit[]exit
DE :R_SE(M3);DE?z;exit[]R_PM(M2);DE?w;exit

上の例で SE の選択オペレータ"[]"の右辺は"exit"オペレータのみになる．PM が "[]" の左辺を実行し，SE が右辺の exit を実行すれば，PM が送信した M1 というメッセージを SE が受理しないことになり，正しい動作が行われない．すなわち，SE が右辺を実行するには，必ず PM も右辺を実行しなければならない．そこで PM が PM?v を実行する場合，PM?v の実行前に"[]"の右辺を選択したことを SE に通知することにする．すなわち上の例に対して次のように個人プロセスを記述すれば，技術者間において正しい動作が行なわれる (S_SE(M4), R_PM(M4) が "A[]B" の右辺を選択したことの通知に関する送受信になる)．

PM:PM?x;S_SE(M1);exit[]S_SE(M4);PM?v;S_DE(M2);exit
SE:R_PM(M1);SE?y;S_DE(M3);exit[]R_PM(M4);exit
DE:R_SE(M3);DE?z;exit[]R_PM(M2);DE?w;exit

いま "A[]B" において A, B の最初のイベントを実行する技術者が，どちらも "p" であるとする．もし "p" が A を選択した場合には，B には含まれるが A には含まれない人に対して，A を選択したことを通知するメッセージを送ることによって正しい実行順序が保たれる (B を選択した場合も同様である)．ただし導出アルゴリズムでは，選択実行プロセス "$A[]B$" において，(1) A, B の最初のイベントを実行する人が同じ "p" であるという制約条件と，(2)A の最後のイベントを実行する人の集合と，B の最後のイベントを実行する人の集合が一致しなければならないという制約条件を課す．これらの制約が満たされない場合，導出はより複雑になる．

5.3 A [> B(割り込み) タイプの記述

"PM?x;DE?y;exit[>DE?z;SE?w;exit" から次のような個人プロセスを導出することにする．

PM:(PM?x;S_DE(M1);exit>>R_DE(M3);exit)
[>(R_DE(M4);S_DE(M4);exit)
SE:(R_DE(M3);exit)[>(R_DE(M2);SE?w;exit)
DE:(((R_PM(M1);DE?y;exit>>(S_PM(M3);exit
|||S_SE(M3);exit))
[>(S_PM(M4);R_PM(M4);DE?z;S_SE(M2);exit))
|[DE?y,S_PM(M4)]|(DE?y;exit[]S_PM(M4);exit))

割り込みのオペレータ"[>"の定義より，DE?z が実行されると，それ以降は "[>" の左辺のイベント系列は実行できない．したがって割り込みを行なう場合には，DE?z の実行を行なう前に DE から，左辺のイベント系列の実行者 (上の例では PM) に対して，割り込みを起こさせるためのメッセージ M4 を送る．PM は M4 を受信することにより割り込みが起こる．その後 PM から DE へ，割り込みを確認した旨のメッセージ M4 を送る．DE が確認メッセージを受信した時点では "[>" の左辺のイベント系列に属する人においては，すべて割り込みが発生している．その後 DE で DE?z を実行する．また "[>" の左辺の最後のイベント DE?y が実行されれば，それ以降は割り込みを起こしてはならず，すべての人は作業を終了しなければならない．このことを実現するため，DE?y の実行後 DE からすべての人に対して作業終了のためのメッセージ M3 を送る．各人は M3 を受信することにより作業を終了する．さらに DE?y の実行後 DE が割り込みを起こさないようにするため，DE?y と S_PM(M4) の一方しか実行できないようにする．ただし割り込み実行プロセス "A[>B" において，A の最後のイベントの実行者 (上の例では DE) は，B の最初のイベント実行者と同じ人でなければならないという制約を課している (他にも若干の制約があるが，詳細は文献 [7] を参照のこと)．

5.4 $A >> B$(順次合成) タイプの記述

A>>B においては，A に含まれるイベントを実行した後に，A の最後のイベントを実行する人から，B の最初のイベントを実行する人に対して，作業終了を知らせるメッセージを送るようにする．それらのメッセージを受理した後に，B のイベントを実行する．

5.5 プロセス呼び出し

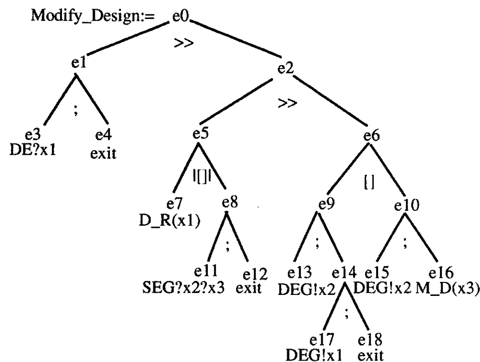
"A=(PM?x;A>>SE?y;exit)[]PM?x;SE?y;exit" というプロセス A 中で A を呼び出しているようなプロセスを考える．

最初 A が呼び出され，選択オペレータ "[]" の左辺が選択された場合，PM?x を実行し新たなプロセス A(A' とする) が呼び出される．A' においても，"[]" の左辺が選択されると，再び PM?x を実行し新たなプロセス A(A" とする) が呼び出される．A" が "[]" の右辺を選択すると "PM?x;SE?y" が実行される．その後制御がプロセス A' に戻り，SE?y が実行される．その後制御が最初のプロセス A に戻り，SE?y 実行される．プロセス A が再帰的に呼び出されると，"$(PM?x)^n;SE?y^n$" が実行される．このプロセス A の個人記述は次のようになる．
PM:A=(PM?x;S_SE(M1);A>>exit)[]PM?x;S_SE(M2);exit
SE: A=(R_PM(M1);A>>y;exit)[]R_PM(M2);SE?y;exit
プロセス A の最初のイベントを実行する人 (PM) が，A を生成する直前に，A に含まれる自分以外の人 (SE) に対して，A の生成を伝えるメッセージ M1 を送る．A に含まれる人は M1 を受信するごとに A を生成する．このように各人でプロセスの呼び出しを同期して行う．

6 導出アルゴリズム

6.1 導出に必要な属性

全体記述から個人のプロセス記述を導出するには，変数やメッセージの送り先などの必要な情報がいる．これらの情報を抽出するため，与えられた全体記述を構文解析し，構文木を作成する．この木の各ノードに，次のような属性を付加させる．以下 3.2.3 のプロセス Modify_Design の構文木 (図 14) を用いて説明する．ただし e0，e1... の添字 0，1... をノード番号と呼ぶ．



*x1...x3，D_R，M_D はそれぞれ次の変数，プロセスを表す
x1:modified_design，x2:result，x3:back
D_R:Design_Review，M_D:Modify_Design
e11 と e12 の間のオペレータ |[]| は |[SEG]| の略
プロセスのゲート名は省略している
図 14.Modify_Design の構文木

1. SP(e):ノード"e" の最初のイベントを実行する人の集合
   属性 SP(e0) の値は {DE} である．

2. EP(e):ノード"e" の最後のイベントを実行する人の集合
   図 14 における属性 EP(e0) の値は {DEG} である．

3. UN(e,w):構文木全体の中で変数"w" を使う人の集合 (UN(e,w) の値は e には無関係)
   図 14 で UN(e1,x1)={SE,SEG,DEG,DE}，
   UN(e2,x2)={SEG,DEG} である．

4. SV(e,w):変数"w" の値を定めるイベントがノード"e" に含まれているとき True，含まれていないとき False となる述語．
   図 14 で SV(e1,x1)=True,SV(e2,x1)=False である．

5. AP(e):ノード e のイベントや変数の受渡しに関与する人の集合，すなわち w1,...,wk が"e" に含まれ，SV(e,wi) の値が True である変数とすると，AP(e) は"e" のイベントを実行する人と構文木全体の中で変数 w1,...,wk を使う人の集合
   図 14 における AP(e1) は，ノード e1 以下で，イベントを実行する人は DE であり，ノード e1 で使われる変数 x1 を使う人は SE，DEG，DE なので，AP(e1)={SE，DEG，DE} になる．

6.2 導出アルゴリズム

与えられた全体記述 P:=B に対する技術者 "p" の個人のプロセス記述を Ip(P:=B) とする．個人プロセスの導出は次のように行われる．

1. 与えられた全体記述 P:=B を構文解析し，B の構文木を作成する．

2. 前述の属性を木の各ノードごとに算出する．

3. Tp(e) を構文木のノード "e" に相当する技術者 "p" の個人プロセス記述とすると，Tp(e) の値を下記のような方法で葉ノードから根ノードへ順次計算し，根ノード e0 における Tp(e0) の値を Ip(P:=B) とする．

以下では，前述のプロセス Modify_Design を用いて，Tp(e) の計算法を説明する．図 14 の構文木において，根ノードである e0 では，順次実行オペレータ">>" によって左右に枝別れしている．5.4 より順次実行では，ノード e の左下にある e1 のイベント中で，最後に実行されるイベントの実行者 (DE) から，右下にある e2 のイベント中で最初に実行されるイベントの実行者 (SE) に対して，作業終了を通知する同期用メッセージを送ればよい．メッセージの送受信者を決めるために，ノード e1 における属性 EP(e1) とノード e2 における属性 SP(e2) を用いる．今，Tp(e1)，Tp(e2) の値が計算されていれば，それらの値と EP(e1)，SP(e2) 等から Tp(e0) の値が計算できる．またメッセージの内容は，ノード e1 のノード番号 "1" を用いる．

次にノード e3 のイベントでは変数 modified_design が使われているので，変数の連絡作業が必要かどうかの判断をしなければならない．ノード e3 の属性 SV(e3,modified_design) が True ならば必要であり，False ならば必要がない．ここでは SV(e3,modified_design)=True なので，変数 modified_design を送信する必要がある．送信すべき人は，ノード e3 の属性 UN(e3,modified_design) でわかる．これらの結果から，Tp(e3) の値が計算できる．

ノード e6 は選択実行のオペレータ ([]) で枝別れしている．5.2 より選択実行にいて，"[]" の左右どちらか一方だけを実行した場合，どちらを選択したかを知らせないと，実行に支障をきたす人が出てくる可能性がある．問題があるのは，選択実行の右辺と左辺での実行者が異なる場合である．図 14 の構文木では，AP(e9)={DEG, DE, SE} で AP(e10)={DEG, DE, SE} なので，この例ではどちらを選択したかを誰にも知らせる必要はない．すなわち Tp(e9) と Tp(e10) が計算されていれば，Tp(e6) の値は計算できる．もし AP(e9) と AP(e10) が異なると仮定すると (AP(e9)-AP(e10)={p} とする)，[] の右辺を選択した時に，"p" にそのことを知らせる必要がある．これらの情報は属性 AP から得られる．

7 あとがき

本報告では，Kellnerの例題をLOTOSを用いて形式的に記述し，その全体記述から，技術者個人のソフトウエアプロセスを自動的に導出した．また導出する方法の概略及び，導出された個人のプロセス記述についての説明を行なった．Kellnerの例題記述全体から，技術者各人のプロセス記述を導出するとサイズが大きくなり，本稿に収まらないので割愛したが，例題のサブプロセスの一部分から導出された個人のプロセス記述においても，他の技術者とのやり取りを頻繁に行なわねばならないので，Kellnerの例題記述の規模においては，個人プロセスを自動生成することは有効である．説明を簡略にするため，PM，DE等の技術者の人数は一人づつに設定したが，これらを複数名にしても，記述のサイズはそれほど変わらない．各個人プロセスの実行は，シミュレータ[16]を用いて行なえる．このシミュレータを技術者の数だけ起動して並列に動かすと，シミュレータ間でメッセージや変数のやり取りが自動的に行なえるので，各技術者は同期用メッセージのやり取りやデータの受渡し操作から解放され，各自の作業に専念できる．類似のLOTOSに関する他のツールについては，文献[10][12]に詳しく説明されている．今後の課題としては，冗長なメッセージを削減する方法の検討がある．例えば，現在のアルゴリズムではプロセスパラメタの値を，そのプロセスに属するすべての技術者に通知しているが，実際には各技術者にとって不必要なパラメタも存在する．そのようなパラメタの通知動作を削除することにより，メッセージの数を削減できる．また現在はゲート名を一人の技術者に対応付けているが，例えばSEがDEとQAEの二人の技術者に対応付ける場合等の導出アルゴリズムについての検討も挙げられる．

参考文献

[1] L. Osterweil, "Software Processes are Software too", Proc.of 9th ICSE pp.2-13, 1987.

[2] M.Suzuki and K. Katayama, "Meta-Operations in the Process Model HFSP for the Dynamics and Flexibility of Software Processes", Proc.of First Int.Conf.on the Software Process, pp.202-217, 1991.

[3] 稲田良造，荻原剛志，井上克郎，鳥居宏次，"ソフトウェア開発過程の形式化とその詳細化による支援システムの作成-ＪＳＤを例として-"，信学論(D-I)，Vol.J72-D-I, No.12, pp.874-882, 1989.

[4] 山口高平，落水浩一郎，"ソフトウェアプロセスモデル構築における知識獲得と利用方式"，人工知能学会，SIG-FAI-9002-3, 1991.

[5] ISO, "Information Processing System,Open Systems Interconnection,LOTOS", IS8807,1989.

[6] M.Saeki,T.Kaneko, and M.Sakamoto, "A Method for Software Process Modeling and Description using LOTOS", Proc.of First Int.Conf.on the Software Process, pp.90-104, 1991.

[7] 東野輝夫，加藤良司，安本慶一，谷口健一，"LOTOSで記述されたサービス仕様からプロトコル仕様群の導出"，信学技報 IN91-111, 1991.

[8] K.I.Kellner,P.Feiler,A.Finkelstein,T.Katayama, L.Osterweil,M.Penedo and H.Rombach, "ISPW-6 Software Process Example",Proc.of First Int.Conf.on the Software Process, pp.176-186 1991.

[9] K.I.Kellner, "Software Process Modeling Support for Management Planning and Control",Proc.of First Int.Conf.on the Software Process, pp.8-28, 1991.

[10] Proc.of 7th and 8th IFIP.WG6,1 Sympo on Protocol Specification, Testing, and Verification, North-Holland, 1987,1988.

[11] 大蒔和仁，二木厚吉，"形式仕様記述言語LOTOSの試用経験"，情報処理，Vol.31.No.10，pp1400-1413，1990.

[12] 高橋薫，神長裕明，白鳥則郎，"LOTOS言語の特質と処理系の現状と動向"，情報処理，Vol.31.No.1.pp35-46, 1991.

[13] R.Gotzhein and G.v.Bochmann, "Deriving Protocol Specifications from Service Specifications Including Parameters", ACM Trans. on Comp.Syst., Vol.8,No.4,pp.255-283, Nov. 1990.

[14] F.Khendek,G.v.Bochmann and C.Kant "New results on deriving protocol specifications from services specifications", Proc. of ACM SIGCOMM'89, pp.136-145, 1989.

[15] R. Langerak "Decomposition of functionality; a correctness-preserving LOTOS transformation", Proc. of Tenth IFIP WG 6.1 Symp. on Protocol Specification, Testing and Verification, North Holland, pp. 229-242, 1990.

[16] 安本慶一，東野輝夫，谷口健一，"LOTOSで書かれたプロトコル仕様群の実行"，信学技報，IN 91-112，1991.