ソフトウェア工学　７８－１
（1991．３．８）

LOTOSによるソフトウェアプロセスの形式的記述法

金子 剛　　川瀬 智　　坂本 正毅　　佐伯 元司
東京工業大学 工学部

ソフトウェアプロセスを仕様記述言語ＬＯＴＯＳで形式的に記述する方法を提案した。本研究では任意のプロセスを形式的に記述するための一方法として、プロセス中で行なわれるタスク (function) と、プロセスに携わる人間 (participants) の二点からなるモデルを提案した。function をリアルタイム構造化分析法に基づいて詳細化し、participants をオブジェクト指向分析法に基づいて詳細化を行なった。また、このモデルを形式的に記述する方法も定型化し、実際の例題記述も行なった。記述言語としては通信機構を記述できる能力を備えている言語ＬＯＴＯＳを採用した。さらに、この例題記述により明瞭になったＬＯＴＯＳの不備な点を補うために、言語の拡張すべき要素を検討した。

A Formal Description Technique for Software Processes in LOTOS

Tsuyoshi Kaneko　　Satoshi Kawase　　Masaki Sakamoto　　Motoshi Saeki
Tokyo Institute of Technology

This paper presents a formal description technique for software processes in LOTOS, which is a formal specification language. Our modeling, called Kardet, consists of two description parts. One is for tasks which are performed in software process and another is for persons, groups, or teams which perfom the tasks. The former is analysed and designed using structured analysis and the latter using Object Oriented Analysis. we have also described the example problem of software process discussed in ISPW6, and discussed the applicability of LOTOS for describing software process.

1 はじめに

ソフトウェア開発プロセスをモデル化し，形式言語で記述すること [1] は，プロセスを実施，理解，ガイダンス，シミュレーションで有用である．

そこで本文ではソフトウェアのプロセスをモデル化するための方法論と形式的仕様記述言語 LOTOS[2] で記述する方法を提案する (図 1)．LOTOS とは，ISO が標準化を進めている仕様記述言語であり，従来，通信プロトコルを主に記述する目的で使われてきた．しかし，LOTOS 言語の持つ可能性は通信プロトコルの記述に留まらず，ソフトウェアプロセスの記述にも適していると考えられる．なぜなら，LOTOS はソフトウェアプロセスに頻繁に生じ得る並列処理，割り込み等を記述する能力を持ち，さらには複数の作業の同期をとりながらの処理を記述する能力があるからである．

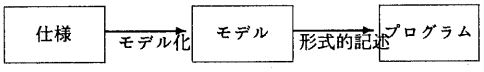


図 1: ソフトウェアプロセスの形式的記述法

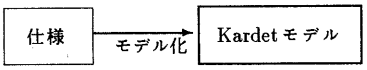
本論の内容は

- このモデルを使い，ソフトウェアプロセスを LOTOS で記述する方法を提示した．
- 実際の例題記述を行なった．
- LOTOS とモデルについて考察を行なった．

となっており，最後にまとめとして今後の課題を検討した．

2 形式記述用モデル

次章に示すソフトウェアプロセスの形式的記述法 Kardet を用いるためのモデル (Kardet モデル) を紹介する．



2.1 モデルの概要

ソフトウェアプロセスの仕様を [いつ], [どこで], [誰が], [何を], [どうやって] という 5 つの視点で分類することは内容を理解し，形式化するのに大変重要である．この 5 つの視点を軸に形式記述用モデル (Kardet モデル, 図 2) を説明していく．

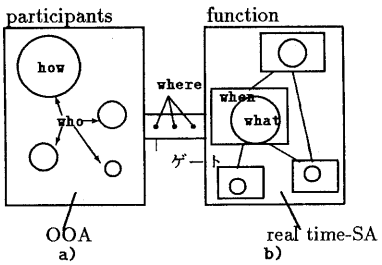


図 2: Kardet モデル

まずソフトウェアプロセスを手順と，それに携わる人という 2 つの視点で大きく分類する．前者を function 部，後者を participants 部と呼ぶことにする．従来のソフトウェアプロセスのとらえ方は手順 (function 部) の behavior にスポットライトが当てられ，それに携わる人や組織 (participants 部) が蔑ろにされていた傾向がある．

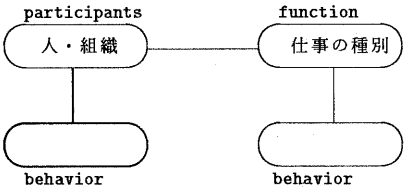


図 3: Kardet モデルの原型

しかし，現実世界では participants に関する behavior，すなわち"人が行なう行為"が存在する．そこで kellner のモデル [4] と異なり，図 3のように participants における bahavior を追加し再構成したものが Kardet モデルである．behavior は人と人のコミュニケーションやふるまいを表している．

このモデルにより，新たにソフトウェアプロセスに関する"手順" の他にプロセスに参加する人間のふるまいを独立した形態で記述できるようになった．例えばプロトコルなど協調モデルを追加できる．従ってプロセスプログラミングで曖昧であった"人間"の管理を記述することができるようになったことは，大いなる成果であるといえよう．

本研究の目的にはソフトウェアプロセスから人々の構成や行動を独立した存在として抽出し，手順と分離することで [誰が] [何を] 行なっているかということをより明確に形式仕様 (プロセスプログラム) に反映させることにある．

図 4: participant のふるまい

それぞれを見てみると participants 部は一つのオブジェクトとして振舞う (図 4)．このオブジェクトは外部からメッセージを受けると作業を行ない，結果を返す．言い替えるとソフトウェアプロセスに於ける作業の種別と作業を行なうのに必要な材料を受け取ると実際の作業を行ないその成果を外部に送り返す役割を担っている．一方 function 部はオブジェクト participants 部が勝手な振舞いをしないように作業を行なうための手順を管理する役割を担っている．Kardet モデルの最上位レベルでは function 部と participants 部が互いに連絡をとりながらソフトウェア開発作業を進めていく，という形に抽象化されている．

そして，function 部の方はリアルタイム構造化分析法 (real time-SA) を，participants 部の方はオブジェクト指向的解析法 (OOA) をベースとする手法をそれぞれ適用しモデル化を行なう．

function 部の内部構造はまず [何を] するのか，どんな作業が要求されているかという視点で仕様を分析し，そこから一つ一つ[1] 切り分けられたタスクがあり，その個々のタスクを [いつ] 開始・終了・再開させるかということを示した．制御構造がある．データの受渡しなどの関係のあるタスクは互いに通信しあっている．(図 2b)

一方 participants 部の中にはソフトウェア開発工程で仕事をしている人や組織を単位としたものが必要な数だけ存在してい

[1] 一つ一つという意味を厳密に定義できなければ形式的記述にはならない．

る．すなわち各 participant はオブジェクトモデルでいうところのオブジェクトである．participants 部の各オブジェクトには自分がどのような作業を行なえばよいかが記述されており，外部からメッセージを受け取ることによって記述中から仕事のやり方を選択し具体的な作業を行なう．各 participant は開発作業を［いかに］行なうかをオブジェクトモデルでいうところのメソッドの形で持っている (図 2a).

participants–function 間で相互に受け渡されるデータ自身は［どこで］扱われるかを明確に認識しており，自分から participants, function の中の作業員もしくはタスクに到達することができる．以上のことに留意してソフトウェアプロセスをモデル化 (図 2) をすれば，最終成果の LOTOS での形式仕様 (プロセスプログラム) は 5 つの視点［誰が］［いつ］［どこで］［何を］［いかに］行なうかが一目瞭然になり，計算機上で実行することも可能である．

2.2 モデルの図表現

実際に 5 つの視点でソフトウェアプロセスを整理するためには表や図で表現することがわかりやすい．適切な表現を検討した結果，Kardet 手法では全部で 8 つの表現形式にまとめることができた．以下の 8 種類の図並びに表はいずれも Kellner[3] の core probrem から作成したものを例として示してある．

2.2.1 data flow chart, mini spec

Kardet モデルでは［何を］行なっているかということはすべてデータの入出力で表現している．したがってプロセスの中で［何を］やるかを data flow chart(図 6) で表す．ただし，一つのタスクに注目した場合，複数の入力すべてを同時に受け取り，すべての出力を同時に行なうわけではない．一部の入力に一部の出力が対応する場合もあるし，実行条件によって出力が選択される場合もありうる．そこで data flow chart を補足するものとして，各作業内容を詳細に記述した mini spec(図 5) を用意する．data flow chart の中で○はファイル，□はタスクを表している．また mini spec の通し番号は作業の順序を規定するものである．

2.4 Schedule and Assign Tasks

1. Requiremant change, Authorizaton を入力し，Project Plans を入力する．そして，Updated Project Plans と Notion of task assignments and scheduled dates と Requiremant change を出力する．

2.5 Modify Design

1. Current design を入力し，Modified Design を出力する．
2. Design review feedback を入力したら，Current design を入力し，Modified Design を出力する．

図 5: mini spec(Core Probrem)

2.2.2 time chart

図 7は LOTOS の並列性をより明確に示すため縦に時間軸をとり，実行する時刻に応じて各作業を配置した．したがって，同じ高さにある作業は同時にできることを表している．これは従来のモデルで用いられていた状態遷移図より，並列的に行なうことのできる作業の存在がわかりやすくなっている．箱の中に書いている数字は core problem での章立ての数字であり，data flow chart のそれと一致する．四角 (□) が作業の開始，四角からの矢印 ( ) が作業中もしくは一旦作業が終了したが再開の可能性があり準備中という意味，矢印の分岐 ( ) は並列処理の発生，これは左に分岐するものも同じ意味を持つ．横線 (–) は作業の消滅，横線からの矢印 ( ) は次の作業を行なう権利が発生したことを示す．作業の消滅とは，プロセスが終了するまで二度と再び再開されないタスクに適用される．この図は並列的に

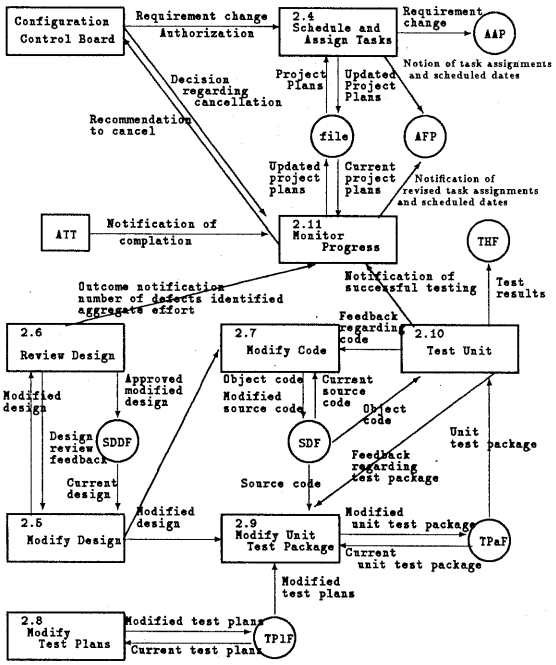


図 6: data flow chart(Core Probrem)

作業が行なわれるのは (2.5, 2.8, 2.11), (2.6, 2.5, 2.8, 2.11), (2.7, 2.8, 2.11),(2.7, 2.9, 2.11), (2.7, 2.9, 2.10, 2.11) という 5 個のフェイズがある．

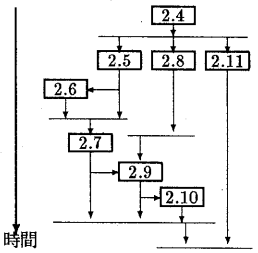


図 7: time chart(Core Probrem)

2.2.3 communication list

ソフトウェア開発工程において，参加者がオンラインで作業する (プログラミングなど) か，文書上で作業する (報告書の提出など) か，口頭で作業を行なう (会議など) か等の記述も重要である．したがって，開発工程中に取り扱われるプロダクトやデータがどのような手段で渡されるかを分類した communication list(図 8) を用意した．この表により具体的に［どこで］データが取り扱われるかが明確に示すことができる．

type	*variable name*
hand (hand carried)	Requirements change Current design Design review feedback Modified design Approved modified design Current test plans Modified test plans
verbal	Authorization Feedback regarding code Feedback regarding test package Decision regarding cancellation Recommendation to cancel

図 8: communication list(Core Probrem)(抜粋)

2.2.4 worker diagram

参加者が data flow chart で規定したどの仕事に関与するかを記述した図が worker diagram(図 12) である．これはプロセスのタスクのうち［どこで］仕事をするかを表した．例えば Schedule and Assign Tasks は Project Manager によって実行される．そして，このタスクでは左に記述してあるように requirements_change と project_plans を入力し，右に記述してあるように updated_project_plans と assignments_and_dates_note を出力する．

2.2.5 participants list

個々の参加者をリストアップし，複数の参加者が誰と組んでチームを構成してどの仕事をするのかをまとめた表である．図 9の参加者は個人としての役割を果たすこともあり，図 10のようなチームを構成して仕事を行なう場合もある．例えば，両方のを見ると参加者 QA Engineer は個人としての QA Engineer, チーム Design Review Team として，チーム Design & QA Engineers としての 3 つの役割があることを示している．

Project Manager
Design Engineer
QA Engineer
Engineer1
Engineer2
Configuration Control Board

図 9: participants list(個々の参加者)

Design Review Team
Design Engineer, QA Engineer, Engineer1, Engineer2
Design & QA Engineers
Design Engineer, QA Engineer
All Affected Personnel
Project Manager, Design Engineer, QA Engineer, Engineer1, Engineer2
All Assigned Personnel
Project Manager, Design Engineer, QA Engineer, Engineer1, Engineer2

図 10: participants list(チームの構成)

2.2.6 activity list

各タスクごとに［どのような］仕事があるか，［いかに］仕事をこなすかを関数としてとらえ，リストアップした表が activity list(図 13) である．例えば，Schedule and Assign Tasks はプランを作成する関数 make_plans をもち，入力 requrements_change と project_plans で updated_project_plans を出力する．

以上で Kardet モデルの 8 つの表現形式を紹介した．

3 ソフトウェアプロセスの形式的記述法 Kardet

前章ではソフトウェアプロセスを 5 つの視点で切りわけ，分類整理するためのモデルを述べた．このモデルを仕様記述言語 LOTOS で記述すれば仕様の形式化が完了する．この記述法 Kardet(図 11) は Kardet モデルを構築する 3 ステップとそれを LOTOS 言語に変換する 4 つのサブステップから構成されている．

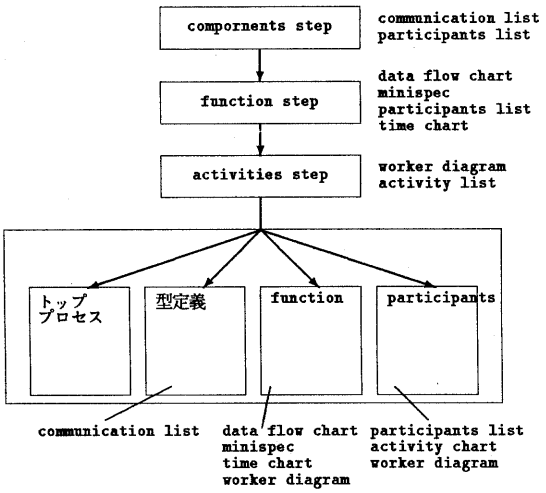


図 11: Kardet

最初の 3 ステップはソフトウェアプロセスを人間が分析し，モデル化するステップである．

3.1 コンポーネント抽出ステップ (components step)

ソフトウェアプロセスの仕様から，開発工程に使用されるファイルや文書が［どこ］にどの媒体を用いて渡されるかを媒体ごとに抽出し，communication list にまとめる．一方，このプロセスに参加する人々を participants list(個人) に列挙する．

3.2 機能詳細化ステップ (function step)

ソフトウェアプロセスから一つ一つのタスクを規定する．タスクの一つ一つとは，単一の作業グループが行なうことが "条件"であり，複数の作業グループが関与する場合は "条件"を満たすまで分解し詳細化する．これらを data flow chart にまとめる．次にもとの仕様と data flow chart から作業 (のみ) を詳細にした spec list を作成する．また，作業グループについては participants list (グループ) を作っておく．

最後にこれらを参考にし各タスクが実行を開始するきっかけとなる開始条件，そのタスクの仕事が二度と再び行なわれることのない完全修了を意味する消滅条件を洗い出し，time chart にまとめる．また，開始や消滅の間にも一旦作業を停止させる中断条件，再び作業が行なわれる再開条件も time chart に導入

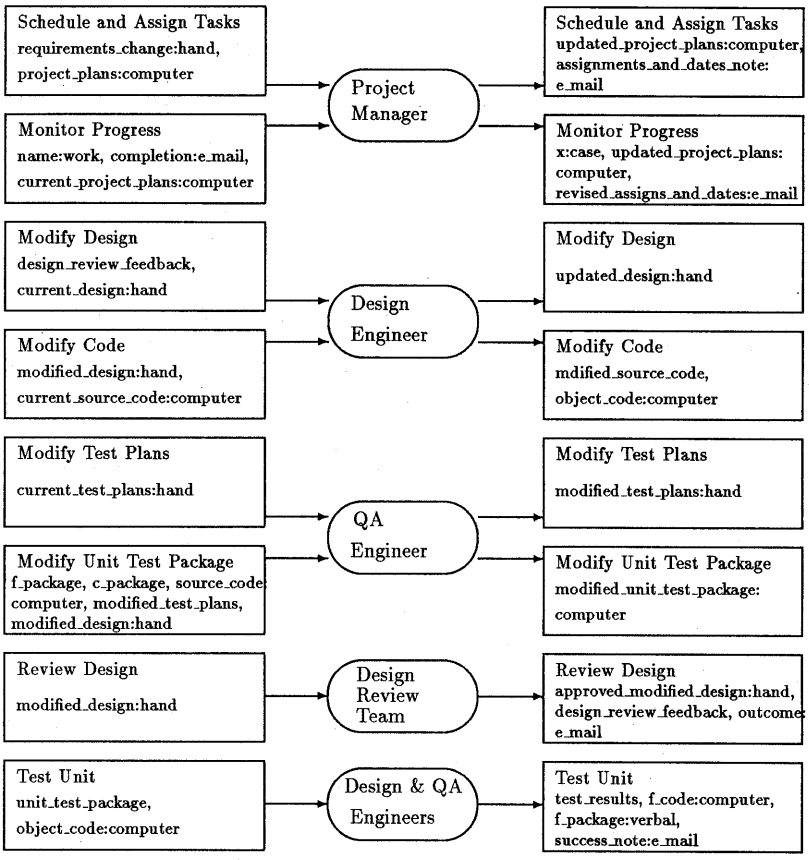


図 12: worker diagram(Core Probrem)

function	Input variable		Output variable	
	name	type	name	type
Schedule and Assign Tasks				
make_plans	requrements_change project_plans	hand computer	updated_project_plans	computer
make_assigns_and_dates	requrements_change project_plans	hand computer	assignments_and_dates	e_mail
Modify Design				
modify_design	current_design design_review_feedback	hand hand	updated_design	hand

図 13: activity list(Core Probrem)(抜粋)

する．以上で function 部に関する 3 つの図とグループに関する participants list を作成した．

3.3 作業詳細化ステップ (activities step)

各作業グループがどのタスクに携わるかを列挙し，worker diagram を作成する．最後に具体的な作業を mini spec と worker diagram から読みとり，activity list を作成する．activity list は，まず仕様に出現する変数のうち参加者が実際に扱うものをタスクごとに分類してその型も同時に列挙しておく．まず、各タスクに関する記述より、そこでどのような仕事をしているのかを分析し，その中の各々の仕事で参加者が何を受け取り、何を返すのかをまとめるわけである．

以上でソフトウェアプロセスを LOTOS で記述するための中間生成物としての 8 つの図表が揃った．

3.4 形式記述ステップ (formal programming step)

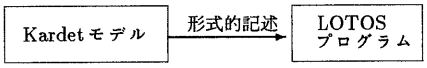


図 14: formal programming step

このステップで 8 つの図表の記述を LOTOS 記述に変換する．

- トッププロセス
- (抽象データ型による) 型定義
- function 部
- participants 部

の 4 つに分けて説明する．LOTOS は大域変数を持たない言語なので LOTOS にファイルシステムのような大域変数の役割を果たす特別な process の定義の仕方を付け加えておく．

3.4.1 トッププロセスの記述法

- ［何を］と［誰が］を表す function 部と participants 部の最上位のプロセス同士をを同期させるための process である．(6，7 行目)
- LOTOS は大域変数を持たない．したがって，全プロセスで利用するようなデータ (例えばファイルのようなもの) も別 process file として定義し，function 部の process と同期させる．(5 行目)

記述の仕方は以下のようになる．

```
 1.specification Core_Problem : noexit
 2.    (* type definitions *)
 3.behavior
 4.  hide pg, fg in
 5.    files[fg] |[fg]|
 6.    participants[pg] |[pg]|
 7.    function[pg, fg]
 8.where
 9.  (* process definitions *)
10.endspec
```

3.4.2 型の定義の仕方

5 つの視点のうち［どこで］を表す部分であり，LOTOS では type に相当する．まず，communication list を基に列挙して定義する．

例

```
type Verbal is

sorts  verbal

opns Feedback_regarding_code,
     Feedback_regarding_test_package,
     Decision_regarding_cancellation,
     Recommendation_to_cancel : -> verbal
endtype
```

次に activity list を関数として定義する．この時リストに入出力の型が 1 種類だけではないような関数があった場合は、変数の型 (e_mail, hand, computer, verbal) を一つにまとめてスーパータイプ all を定義する．

3.4.3 function の記述法

以下の手順で行なう．

1. 制御，スケジューリング (例えばタスク A が終了してからタスク B を始めるなど) に関する部分とタスク本体を分離し，同期させる．

```
process function[cg, pg, fg]:exit
:=
  hide sg in
    task[pg, fg, sg] |[sg]| schedule[sg]
endproc
```

2. time chart の各タスクの開始 (wake)・消滅 (sleep) 条件などから，process schedule を作る．下の数字 4, 5, 8, 11 はいずれもタスク ID であり，time chart の番号 2.4,2.5,2.8,2.11 に対応する．

```
process schedule[sg] : exit
:=
sg !4 !sleep; (sg !5 !wake; schedule[sg] |||
               sg !8 !wake; schedule[sg] |||
               sg !11 !wake; schedule[sg])
[]
.........................
endproc
```

3. data flow chart からタスクの相互関係を読みとり，ゲートでそれぞれを連結する．

```
process task[pg, fg, sg] : exit
:=
  タスク 1[pg, fg, sg] || タスク 2[pg, fg, sg]
  || ................
endproc
```

4. 各タスクは以下のように記述する．
   - worker diagram で示された作業員をゲートで指名し，作業に必要な入力データ，ドキュメントもゲート pg を通して送る．(pg !design_engineer !review_design; pg !c_design;) これは，design_engineer にタスク review_design を割り当てた後，必要となるドキュメント c_design を送る例である．
   - タスクの成果は各タスク process に戻されなくてはいけない．これは，participant から成果をゲート pg を通して受け取るという形で記述する．(pg ?u_design:hand;)
   - タスクの成果をそれを必要としている他のタスクへゲートを介して渡す．(outg !u_design; g11 !complation;)

- communication list に記述されている各タスクの終了条件を満たせば，くりかえし，つまり自己再帰を行なうように，消滅条件を満たせば，exit でプロセス自身が消滅するように書く．例えば，process Modify_Design は schedule process よりゲート sg を介して指示があれば消滅する．
  (sg !5 !sleep; exit [] Modify_Design[pg, sg, fg, ing, outg, g11])

これらのステップをまとめると以下のようなタスク process Modify_Design が出来上がる．

```
process Modify_Design[pg, sg, fg, ing, outg, g11]
:exit:=
  ( sg !5 !wake;
    fg !SDDF !read ?design:hand; exit []
    ing ? c_design:hand; exit)>>
   pg !design_engineer !review_design;
   pg !c_design;
   pg ?u_design:hand;
   outg !u_design; g11 !complation;
  ( sg !5 !sleep; exit []
    Modify_Design[pg, sg, fg, ing, outg, g11] )
endproc
```

この記述法を使うと function は全体として，以下のように記述される．

```
process function[pg]
     トッププロセス [pg, sg] |[sg]| schedule[sg]
where
  process schedule[sg]
      タイミング管理
  endproc
  process トッププロセス [pg, sg]
      タスク同期
  where
     process 各タスク
          作業内容
    endproc
    ......................
  endproc
endproc
```

3.4.4 participant の記述法

以下の手順で行なう．

1. participants list にある各作業グループを LOTOS の process とし，それらをチョイスオペレーター ([]) で繋いだものが最上位 process participants のふるまいとなる．

```
process participants[pg]: noexit
:=
  hide g1, g2 in
        Project_Manager[pg]            []
        Design_Engineer[pg, g1, g2]    []
        QA_Engineer[pg, g1, g2]        []
        Design_Review_Team[pg, g1]     []
        Design_and_QA_Engineer[g2]
where
   ...............................
endproc
```

2. グループ中の各構成員を LOTOS の process とし，その構成員同士の連絡用のゲート (g1 or g2) を決め同期させる．以下の例はグループ中の構成員 process Design_Engineer と QA_Engineer を同期させた例である．この両者はゲート g2 で通信を行なう．

```
process Design_and_QA_Engineers[pg, g2]: noexit
  :=
    hide g1 in
      Design_Engineer[pg, g1, g2] |[g2]|
      QA_Engineer[pg, g1, g2]
  endproc
```

3. 構成員 process の記述は以下のようにする．

- worker diagram より，その構成員 process がどのタスクを実行するかを抽出し，記述する．例えば，design_review_team は review_design を実行する．(pg !design_review_team !review_design;)
- activity list に示された，作業に必要なデータを受けとる部分を記述する．(pg ?modified_design:hand;)
- 構成員間で行なわれる通信を詳細化する*．(ask[g1] [] do[g1])(後述)
- activity list に示された関数で計算し，その値をゲート pg でタスクに返すように記述する．．(pg !approve_design(modified_design)

以上のふるまいを繰り返すように再帰な構造にする．

例

```
process Design_Engineer[pg, g1, g2]: noexit
:=
  (pg !design_review_team !review_design;
   pg ?modified_design:hand;
   (ask[g1] [] do[g1]);
   pg !approve_design(modified_design);
   Design_Engineer[pg, g1, g2])
  []
  (pg !design_engineer ?x:work;
  ([x = modify_design] ->
       (pg ?current_design:hand
           ?design_review_feedback:hand;
        pg !modify_design(current_design,
                 design_review_feedback);
        Design_Engineer[pg, g1, g2])
  []
 ...............................
endproc
```

上記の "通信の詳細化*" ステップはチームとしての作業を行なわずに個人のみで行なう時には省略される．(上記の例の [x = modify_design] ->以降)

本研究で記述したプログラムではプロトコルとして David A. Marca が考案した Negotiation Protocol を用いて記述してある．Negotiation Protocol は，一つの要求に対し尋ねる人 (ask) と答える人 (do) が，お互いに意見を交わしながら一つの結論を導き出していくという折衝をプロトコルという形でまとめたものである．ただし，プロトコルの箇所は negotiation protocol を用いる必然性はなく，プロセスの正確を反映しているものであるなら他のプロトコルでも構わない．

この記述法を用いると participants 部は以下のようになる．

```
process participants
       グループチョイス
where
    process グループ名
        構成員 1 [] 構成員 2 [] .......
     endproc
     .......................
    process 構成員
        negotiation
    where
       process negotiation
            折衝
       endproc
    endproc
    .......................
endproc
```

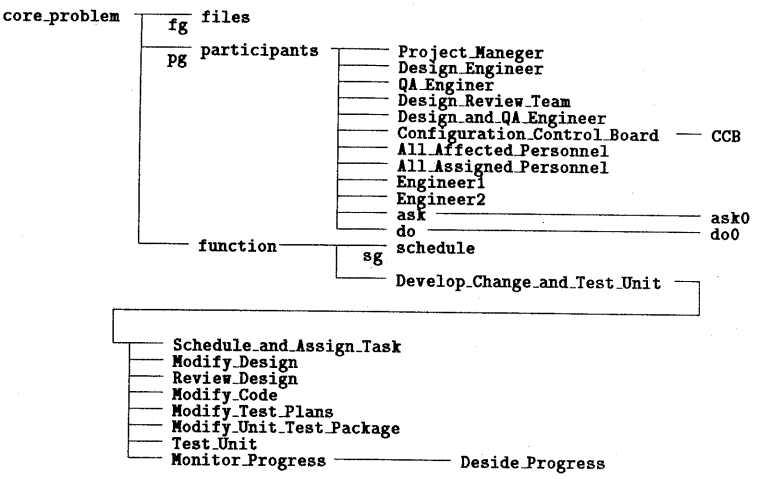


図 15: 形式記述における process の構成図 (core problem)

3.4.5 files の記述

今回出てきた大域変数はファイルシステムのみである．これは以下のように記述される．

```
process files[fg]
(file:computer, sddf:hand,
 tpaf:computer, thf:computer)
: noexit :=
  (((fg!FILE!read!file; exit []
     ...............................
 ) >>
    files[fg](file, sddf, tpaf, thf)
   ) [] (
    fg!FILE!write?x:computer;
    files[fg](x, sddf, tpaf, thf) []
   ........................................
   )
  )
```

おのおののファイル (FILE, SDDF, SDFS など) について読みだし部 (read) と書き込み部 (write) を設定し，入力もしくは出力が終ったら，ファイルという process を再帰させることでファイルシステム自身は初期化される．具体的にファイルゲート fg で受け渡されるデータは

fg!ファイル名!オペレーションの種類 (!or?) ファイルの内容

という記述になっている．オペレーションの種類は read か write で，データの 3 番めのファイルの内容が定数になっていれば読みだし，変数になっていれば書き込みである．ファイルシステムはすべてこの記述法で設定できる．

以上の記述法により作成された LOTOS プログラムは，participants 側はオブジェクト指向スタイルに，function 側はイベント指向スタイルに記述される．

4 実際の形式記述例

ISPW6 で比較検討されたソフトウェアプロセスの一例として Marc Kellner の例題; SOFTWARE PROCESS MODELING EXAMPLE PROBLEM[3] を記述した．この例題は core problem と optional extension とに分かれているが，本論での記述は core problem のみを扱った．その結果テキストファイルで 7００行２３ K バイトの大きさになった．そのモジュール構造を図 15に示し，以下に基本部分のみのコードを掲載する．

```
specification Core_Problem
(file:computer, sddf:hand, sdfs:computer, sdfo:computer, tplf:hand,
 tpaf:computer, thf:computer, requirement_change:hand)
: noexit

(* type definitions *)

behavior
  hide pg, fg in
   files[fg](file, sddf, sdfs, sdfo, tplf, tpaf, thf) |[fg]|
   participants[pg](requirement_change) |[pg]|
   function[pg, fg]
where

process function[cg, pg, fg]:exit
:=
  hide sg in
    Develop_Change_and_Test_Unit[pg, fg, sg] |[sg]| schedule[sg]
where
  process schedule[sg] : exit
  :=
  sg !4 !sleep; (sg !5 !wake; schedule[sg] |||
                 sg !8 !wake; schedule[sg] |||
                 sg !11 !wake; schedule[sg])
  []
  sg !6 !sleep; sg !5 !sleep; sg !7 !sleep; schedule[sg]
  []
  sg !10 !sleep; sg !7 !sleep; sg !9 !sleep; schedule[sg]
  []
  sg !8 !sleep; sg !9 !wake; schedule[sg]
  []
  sg !11 !sleep; exit
  endproc

process Develop_Change_and_Test_Unit[pg, fg, sg] : exit
:=
   (Schedule_and_Assign_Tasks[pg, sg, fg]                          ||
    Modify_Design[pg, sg, fg, g6-5, g5, g5-mp]                     ||
    Review_Design[pg, sg, fg, g5, g6-5, g6-11, g6-mp]              ||
    Modify_Code[pg, sg, fg, g5, g10-7, g7-mp]                      ||
    Modify_Test_Plans[pg, sg, fg, g8-mp]                           ||
    Modify_Unit_Test_Package[pg, sg, fg, g5, g10-9, g9-mp]         ||
    Test_Unit[pg, sg, fg, g10-7, g10-9, g10-11, g10-mp]            ||
    Monitor_Progress[pg, sg, fg, g6-11, g10-11,
                     g5-mp, g6-mp, g7-mp, g8-mp, g9-mp, g10-mp]
   )
where
   process Schedule_and_Assign_Tasks[pg, sg, fg]: exit
   :=
     pg !CCB !authorization ?requirements_change:hand;
     fg !FILE !read ?c_plans:computer;
     pg !project_maneger !schedule_and_assign_tasks;
     pg !requirements_change !c_plans;
     pg ?n_plan:computer !schedule:e_mail;
     pg !all_affected_parsonnel !schedule;
     pg !all_assigned_parsonnel !requirements_change;
     fg !FILE !write !n_plan;
     sg !4 !sleep; exit
   endproc

   process Modify_Design[pg, sg, fg, ing, outg, g11]:exit
   :=
```

```
    ( sg !5 !wake; fg !SDDF !read ?design:hand; exit []
      ing ? c_design:hand; exit)>>
     pg !design_engineer !review_design;
     pg !c_design;
     pg ?u_design:hand;
     outg !u_design; g11 !completion;
    ( sg !5 !sleep; exit []
      Modify_Design[pg, sg, fg, ing, outg, g11] )
  endproc

  process Review_Design[pg, sg, fg, ing, outg5, outg11, g11]:exit
  :=
    ing ?design:hand;
    pg !design_review_team !review_design;
    pg !design;
    pg ?outcome:e_mail;
    ([outcome = unconditional_approval] ->
        pg ?design:hand; fg !SDDF !write !design; exit
     []
     [outcome = minor_change or outcome = major_change] ->
        pg ?design:hand; outg5 !design; exit
    )>>
    outg11 !outcome; g11 !completion;
    ( sg !6 !sleep; exit []
      Review_Design[pg, sg, fg, ing, outg5, outg11, g11])
  endproc

 ..............................................................

  process Monitor_Progress
          [pg, sg, fg, ing6, ing10, g5, g6, g7, g8, g9, g10]
  : exit :=
    sg !11 !wake; fg !FILE !read ?c_plans:computer;
    (( g5 ?completion:e_mail;
       pg !project_maneger !monitor_progress;
       pg !modify_design !completion !c_plans; exit
       []
       g6 ?completion:e_mail;
       pg !project_maneger !monitor_progress;
       pg !review_design !completion !c_plans; exit
       []
        ........................................
       []
       ing6 ?outcome:e_mail;
       pg !project_maneger !c_plans !outcome; exit
     )>>
      pg ?y:case;
      ( [y = severe] -> pg !CCB !cancel; pg !CCB ?decision:verbal;
                        ( [decision = cancel] -> exit
                          []
                          [decision = resumpt] ->
                                pg !project_maneger !decision;
                                Decide_Progress[pg, fg]
                        )
       []
       [y = deviate] -> Decide_Progress[pg, fg]
      )
    []
     ing10 ?success:e_mail; sg !11 !sleep; exit
    )
  where
    process Decide_Progress[pg, fg]: exit
    :=
      pg ?u_plans:computer ?schedule:e_mail;
      fg !FILE !write !u_plans;
      pg !all_affected !schedule; exit
    endproc
  endproc(* Monitor_Progress *)
endproc(* function *)

process participants[pg](requirements_change : hand) : noexit
:=
hide g1, g2 in
Project_Manager[pg]
[]
Design_Engineer[pg, g1, g2]
[]
QA_Engineer[pg, g1, g2]
[]
Design_Review_Team[pg, g1]
[]
Design_and_QA_Engineers[pg, g2]
[]
Configuration_Control_Board[pg](requirements_change)
[]
All_Affected_Personnel[pg]
[]
All_Assigned_Personnel[pg]

where
process Design_Review_Team[pg, g1] : noexit
:=
hide g2 in
Design_Engineer[pg, g1, g2]
|[g1]|
QA_Engineer[pg, g1, g2]
|[g1]|
Engineer1[pg, g1]
|[g1]|
Engineer2[pg, g1]
endproc

 ........................................

process Project_Manager[pg] : noexit
:=
(pg !project_manager ?x : work;
 [x = schedule_and_assign_tasks] ->
       (pg ?requirements_change : hand ?project_plans : computer;
        pg !make_plans(requirements_change, project_plans)
           !make_assigns_and_dates(requirements_change, project_plans);
        Project_Manager[pg])
 []
 [x = monitor_progress] ->
       (pg ?name : work ?completion : e_mail ?current_project_plans : computer;
        pg !result(monitor_progress(name, completion, current_project_plans),
                   name, completion, current_project_plans);
        pg !update_plans(x, completion, current_project_plans)
           !update_assigns_and_dates(x, completion, current_project_plans);
        pg !all_affected !assignments_and_dates;
        Project_Manager[pg]))
[]
(pg !all_affected_persons ?assign_dates_note : e_mail;
 Project_Manager[pg])
endproc  (* end Project_Manager *)

process Design_Engineer[pg, g1, g2] : noexit
:=
(pg !design_review_team !review_design;
 pg ?modified_design : hand;
 (ask[g1](modified_design, 0, review_design)
  []
  do[g1](modified_design, 0, review_design))
 >> accept outcome : e_mail in
 pg !outcome;
 pg !check(outcome);
 Design_Engineer[pg, g1, g2])
[]
(pg !design_and_qa_engineers !test_unit;
 pg ?object_code:computer ?unit_test_package:computer;
 (ask[g2](object_code, unit_test_package, test_unit)
  []
  do[g2](object_code, unit_test_package, test_unit))
 >> accept test_results : computer in
 pg !test_results;
 Design_Engineer[pg, g1, g2])
[]
(pg !design_engineer ?x : work;
([x = modify_design] ->
       (pg ?current_design : hand ?design_review_feedback : hand;
        pg !modify_design(current_design, design_review_feedback);
        Design_Engineer[pg, g1, g2])
 []
 [x = modify_code] ->
       (pg ?modified_design : hand ?current_source_code : computer;
        pg !modify_to_source(modified_design, current_source)
           !modify_to_object(modified_design, current_source);
        Design_Engineer[pg, g1, g2])
[]
(pg !all_affected_persons ?assign_dates_note : e_mail;
 Design_Engineer[pg, g1, g2])
endproc (* Design_Engineer *)

 ..............................................................

endproc (* end participant *)
endspec
```

5 考察並びに評価

5.1 Kardet モデルに関する評価

5.1.1 プロセスの実行

Kardet を用いて記述したプログラムは部分実行しても実プロセスに有効である．たとえば，participants 部 のみを計算機上で実行した場合，function 部がないので各タスクで一体何をやっているのか．作業に必要な材料は一体何なのかを要求し，実行が一時中断する．オペレータがこういった材料を入力すると participants 部は再び動きだし，次のフェイズで再び必要な材料を聞いてくる．このように部分実行すると function 部でいったい何がやられていたかが明らかになる．逆に function 部を実行すると perticipants で何がやられていたか，すなわち誰がどんな作業をやっていたかが明らかになる．この利点は，人とプロセスを分離してモデル化した Kardet モデル特有の効果である．

5.1.2 プロセスの追加変更

ソフトウェアプロセスにマイナーチェンジの技法がまだ確立されていないことが Kardet の今後の課題として残ってしまった．また LOTOS 言語自身にも process の生成消滅ができるように拡張をする方が良い．

5.2 Kardet を Kellner の例題に適用した効果

ソフトウェアプロセスを形式的に記述するとプロセスにおける矛盾点，不備な点があぶり出されてくるという利点がある．kellner の例題でも Kardet で形式記述した時に様々な曖昧性が仕様に残されていることに気がついた．

5.3 言語 LOTOS に関する評価

5.3.1 記述能力の限界並びに言語の拡張

[5] で LOTOS の記述スタイルに関する例が示されている．ここにはごく簡単な一つの仕様を Monolithic style, Constraint oriented style, State oriented style,Resource oriented style で記述した例がある．このように様々な問題を記述するための方法論，記述スタイルの研究成果も上がってきている．

また，ソフトウェアプロセスプログラミングの目的として，Enaction と Simulation があることは前述した．Enaction とはプロセスプログラムにのっとって実プロセスを起動することであり，Simulation とは疑似プロセスを計算機上で実行させることである．LOTOS 言語の性質上インターリービングで実世界のパラレルな作業を疑似的に記述しているので Enaction を目的としたプログラムではない．本論文のプログラムも Simulation を主目的としている．

ある participant が作業中に他の仕事を受けとることがある．なぜなら function 部は (疑似) 並列作業を許しており，各 participant も個人としての作業，グループとしての作業の 2 種類の役割を持っている．したがって場面によっては一人に二つ以上のタスクが衝突する可能性がある．これを解消するためには participant 間の関係を記述した部分がプログラムの中になければならない．この関係は Kardet モデルに E-R モデルを導入し，LOTOS 自身も E-R モデルを記述できるように拡張すれば良いと思う．

5.3.2 オブジェクト指向性

言語 LOTOS をオブジェクト指向的に取り扱おうとするアプローチは [6][7] など，意欲的に取り組まれている．オブジェクトモデルのオブジェクトを process に，メソッドを behavioral expression に置き換え，メッセージの受渡しはゲートを介して通信させることにより，オブジェクトモデルを LOTOS で記述しようとする試みである．この段階まではオブジェクト指向モデルを LOTOS で記述することが可能である．

実際に Kellner の例題でクラスとインスタンスの概念が欲しいと思ったところは 2 ヶ所ある．一つ目は，Review_Design において同じ動きをする participant が二人いた．(仕様では two other software engineers とある) この二人をそれぞれ独立したものとして区別するなら，(例えば番号を 1，2 とかふるなら) クラス化した方がよい．区別しないのならプロセス software engineer を 2 回呼ぶだけでよい．5 章のプログラムではどちらの方式もとらず，同じ内容の process を Engineer1,Engineer2 として記述した．しかし，同じ内容のオブジェクトが数多く存在する場合一々独立な process として記述する場合，大変に効率が悪い．そこで，代表的な process をクラスとして定義して，必要な数だけインスタンスを生成できるようにしたい．

もう一つは，複数のプロセスに共通の性質が Global Issues として仕様に記述してあった．これは全プロセスに共通の入出力が書いてあっただけのものだが，実際にプログラミングする時に全 process にいちいち同じことを書いていては非効率的と考える．そこで，Global Issue という process を一つ作り，その性質を全 process に継承させるようにすれば，記述量が減る．

インスタンスの生成が実現できれば，ソフトウェアプロセスのマイナーチェンジが行なわれた時にも有効に応用でき，仕様の変更問題を解決することができる．

6 おわりに

本論文では，プロセスプログラミングの研究の中でも形式的な記述法に焦点を絞り Kardet を紹介した．Kardet を使用するためのより一般的なモデルを紹介し実際のソフトウェア開発工程を Kardet モデルを用いて言語 LOTOS で形式的に記述した．Kardet モデルは手順と人を分離するという新たな視点が導入されている．一石を投じることになった．今後もこの Kardet モデルを基に分析・記述を進め，その適応性を評価していきたい．

参考文献

[1] Osterweil,L.,"Software Processes are Software Too", Proceedings of the 9th,pp.2-13,Mar.-Apr.1987

[2] "Infomation processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour",ISO 8807.

[3] Marc Kellner, "SOFTWARE PROCESS MODELING EXAMPLE PROBLEM", 9th. Aug. 1990.

[4] Marc I.Kellner, Gregory A.Hansen "Software Process Modeling: A Case Study", Proc.of. 22th HICSS pp.175-188(1989)

[5] ISO/IEC JTC1/SC21 N3067,"Specification styles for structuring of OSI Formal Descriptions",19 Sep. 1988

[6] Thomas Mayr,"Specification of Object-Oriented Systems in LOTOS", Formal Description Techniques, pp.107-119,Elsevier Science Publishers B.V. (North-Holland),1989

[7] Kilsup Lee, Jin Ho Hur,"O-LOTOS: An Object-Oriented Extension of LOTOS for Distributed Processing",SG-ODP K116, Aug.24,1989.