ソフトウェア工学 83−12
（1992. 2. 7）

ソフトウェアプロセスにおける協調メカニズムの抽象化について

松浦 佐江子　　本位田 真一
情報処理振興事業協会

ソフトウェアプロセス記述についてさまざまなモデリング技法や記述言語の研究が行なわれている。プロセスを記述する目的は多様であるが、本稿では、プロセスやノウハウの再利用や要求を仕様化するためのガイドラインと考える。ここで問題となるのは再利用をするためにどのようなモデリングや記述が望まれるかということである。プロセス記述に限らず、ソフトウェアの仕様化のプロセスにおいては具体的な問題に特化して仕様を構築するために仕様の部品化や再利用が難しくなっている。そこで、仕様の構築において抽象化・具体化の観点が重要であると考える。例えば、設計技法を形式的に記述し、個々の技法の性質を一旦抽象化して、それを他の状況に適合させることによって、1つの仕様化プロセスを複数の技法に分担させるといった構築法が考えられる。また、ソフトウェアプロセスの記述においてはタスクとリソースの協調動作が重要なポイントであり、こうした協調メカニズムの抽象化を行なうことが再利用をするためには必要である。本稿では、Kellner の例題を用いて Miranda によるソフトウェアプロセスの記述実験を行ない、ソフトウェアプロセスにおける協調メカニズムとその抽象化について考察する。

Abstraction for Cooperative Mechanism on Software Processes

Saeko Matsuura　　Shinichi Honiden
Information-technology Promotion Agency
Shuwa-shibakoen 3-chome Bldg. 3-1-38 Shibakoen, Minato-ku, Tokyo 150 Japan

Several approaches to software process modeling and describing are studying. There are various reasons why we study on software process. In this paper, we consider its reson as reusing concrete software processes and know-how on them and using them as a guideline on specify a given requirement. Then it is important how to construct its model or describe it for the purpose of reusing software processes. In other cases , it is also difficult for us to construct reusable software components for the reason of that it is easier for us to specify a give requirement concretely responding to its features. So we think that an abstract and concrete view is important to construct a specification. For example, the following specify process is considered. First we write some design methods using formal languages , then change each one into abstract state by extracting their characters. In the next step we will adapt them to another concrete situations and get concrete description of them. It enables us to take partial charge of a specifying process to several methods. On the other hand , it is one of the key points to describe a software process how we write its cooperative behavior between its tasks and resources. We think that it is necessary to abstract such behaviors for reusing software processes. In this paper we report an experimental description of Kellner's problem using functional language Miranda. And we discuss about cooperative mechanism on software process and thier abstraction.

1 はじめに

ソフトウェアプロセス記述についてさまざまなモデリング技法や記述言語の研究が行なわれている [8]。プロセスを記述する目的は多様であるが、本稿では、プロセスやノウハウの再利用や要求を仕様化するためのガイドラインとしての利用を考える。

ここで問題となるのは再利用をするためにどのようなモデリングや記述が望まれるかということである。プロセス記述に限らず、ソフトウェアの仕様化のプロセスにおいては具体的な問題に特化して仕様を構築するために仕様の部品化や再利用が難しくなっている。そこで、仕様のもつ性質を抽象化し、それを他の具体的な状況に適合させるといった手段が必要であると考える。例えば、こうした考えに基づいてつぎのような仕様の構築法が考えられる。まず、設計技法を形式的に記述し、それらを仕様化プロセスにおける最適な場面に適用すること、すなわち仕様の合成を考える。この場合に、設計技法のもつ抽象的な性質をうまく利用する必要がある。そこで、個々の技法の性質を一旦抽象化して、それを他の状況に適合させることによって、1つの仕様化プロセスを複数の技法に分担させる。

このようなことから、プロセスやノウハウを再利用するためには、プロセス記述言語としては解析可能な形式的記述ができることと、先に述べた他の具体的な状態に適合したプロセスやノウハウを導出するための手段を持つことが必要であると考える。

また、ソフトウェアプロセスの記述においては各タスクとリソースの関係の定義およびそれらの相互作用による協調動作が重要なポイントであると思われる。こうした協調メカニズムの個々の状況によらない抽象化を行なうことが再利用をするためには必要である。

こうした抽象化を考察するために関数型言語に着目する。関数型言語は関数抽象と関数適用という単純な枠組と型をもつ言語である [2]。特に Miranda は [4] 無限データを扱えることから、ソフトウェアプロセスのような並列プロセスの動作記述が可能であると考える。この関数型の特徴を活かしてソフトウェアプロセスにおけるインタラクションの定式化と環境適合的な枠組の構成を実験したので、これを報告する。2節においてソフトウェアプロセスの記述の視点と関数型言語 Miranda の特徴およびプロセス記述への適用の観点を述べる。3節において、ソフトウェアプロセスワークショップで提案された Kellner の問題を例題として、この記述について報告する。4節においては、他の仕様記述言語との比較を行なう。最後にインタラクションの定式化とその抽象化・具体化によるプロセスの利用について述べる。

2 関数型言語によるモデリング

2.1 ソフトウェアプロセスの記述

Kellner の例題 [3] はソフトウェアプロセスのモデリングの方法の比較を行なうことを目的としているが、その中でも述べているように、モデリングの方法も、そのゴールとする目的が何であるかによって比較の観点が異なってくる。本稿では、ソフトウェアプロセスにおけるプロセスやノウハウの再利用を行なうための、プロセスにおけるプリミティブの記述を目的とし、その抽出に対して関数型言語の果たす役割を考察する。

ソフトウェアプロセスの特徴をつぎのように考える。

- 非同期プロセスの集合としてモデリングされる。
- 各プロセス間にさまざまな相互作用があり、こうした協調メカニズムが重要である。

ソフトウェアプロセス記述言語として JSP や LOTOS などがしばしば取り上げられている。LOTOS は並列性、非決定性、同期・非同期の相互作用、中断といった記述の枠組を持っている。一方、次項で説明する Miranda ではストリーム処理関数のネットワークとして並列プロセスの関連を表現できる [1]。それに加えて、高階関数を使った記述等、関数という単位による抽象的な記述を可能とする。ただし、プロセス記述のシミュレーションとしては非決定性のスケジューリングを陽に与える必要がある。

本稿ではプロセス記述の再利用という観点で Miranda による記述を実験する。

2.2 Miranda

Kent 大学の Tuner によって設計された関数型言語 Miranda [4][7] は、Hope,ML[6] といった関数型言語の流れを汲む、多相型・型推論・抽象データ型・代数的型定義を取り入れた非正格な言語である。そして遅延評価機構によって、ストリームを扱うことができる。

前項で述べたように、プロセスの記述言語においては各プロセス間における相互作用をうまく記述したり、その枠組を再利用することを考える必要がある。そこで、Miranda を採用した理由は以下のとおりである。

- 関数抽象と関数適用という単純なメカニズムであり、1つの関数が非同期プロセスを表現できる。

すなわち、関数抽象によって1つのプロセスを定義し、関数適用によって他のプロセスとのデータの授受を行なう。データとして関数自身を扱うことができる。

- 計算対象がすべて型をもつことで関数の意味が陽に定義できる。豊富な型定義が行なえる。
- 遅延評価機構によるストリームの表現が可能であるので、ストリームの処理関数のネットワーク方程式として非同期プロセス間の相互作用をモデル化し、システムの並列動作を記述できる。
- 実行可能である。

2.2.1 プロセス記述における視点

関数型言語においては"関数"が基本単位である。関数型言語におけるプロセスの考え方はつぎのようになる。関数Fの定義において関数G、Hが参照されているとする。

F(A,B) = ...G(A).....H(B)

個々の関数がプロセスを構成すると考えられる。右辺で他の関数を参照している場合は、他のプロセスからの送信を受けていることになっている。すなわち、LOTOSにおける"ゲート"は関数の定義内部に表現される。参照されている関数においてはその入力が参照している関数からのデータの受信になっている。例えば、上記の関数において、プロセス"F"はプロセス"G"とプロセス"H"と通信しており、A,B,G(A),H(B)がゲートにおいて授受されるデータである。A,Bがプロセス"G"とプロセス"H"に送られるデータであり、G(A),H(B)が、プロセス"G"とプロセス"H"からプロセス"F"に送られるデータであると考えられる。

プロセス間で授受されるデータの系列をストリームと見なす。システム内で発生する事象をストリームの要素と見なすことができる。

高階関数の概念は、プロセスがプロセスの授受を行なうことができることを意味する。これをプロセスの起動に用いる。

2.2.2 インタラクション

非同期プロセスとは、並列に実行されるプロセスのことである。これらの相互作用は同期(synchronization)あるいは通信(communication)によって規定される。関数の機構におけるプロセスの考え方は前記のようになっている。すなわち、関数は1つの非同期プロセスを表している。この時データをどのように評価するか、すなわち評価機構が問題になる。Mirandaは遅延評価を行なうので、これによって上記の関数のゲートにおける同期が表現される。

LOTOSの場合はプロセスの振舞いをCCS(Calculus of Communicating Systems)に基づいて記述する。すなわち、各独立したプロセスの相互関連を記述するための枠組が用意されている。

一方、Mirandaは、関数の適用によりプロセス間のインタラクションを表現するので、CCSのような枠組を記述者が管理する必要がある。しかし、これが定式化できれば、1つのプロセス記述のテンプレートとして活用できる。こうした枠組を使って並行動作をストリームの処理関数のネットワークとして記述すると、遅延評価機構によって、各プロセスが互いに同期を取りながら動作する。ここで、通信はストリームに出力したデータによって行なわれると考える。

3 例題の記述

3.1 例題の分析

この例題はあるプロジェクトによるソフトウェアの開発が行なわれ、すでにプロダクトがある状態を仮定している。この時、変更の要求によって生じるソフトウェアプロセスを記述することが目的である。各変更のプロセスに対する要求が自然語で記述されている。要求は、各変更のプロセスを単位として、概要・入力や出力のデータとデータ授受の相手・プロセスの責任者・制約として記述されている。

そこで、以下のような項目を考察しながらプロセス仕様を記述する。

- プロセスの記述
  各プロセスのタスクを分析し、それぞれのタスクを定義する。さらに、それらの同期を考慮してプロセスを作業の列として表現する。この時、時間的順序が規定されている作業と、並列に実行されうる作業とを区別する。ただし、並列に実行されうる作業とは順序を考慮しなくてよいということである。他のプロセスの起動、タスクのインターリーブを基本メカニズムとして定義する。
- プロセス間の同期
  各プロセスの出力をストリームとして束縛し、それを参照することで同期を記述する。すなわちそ

のストリームを入力とすることは、対応するプロセスの出力が有効になった時点でそのタスクが実行されることを意味することになる。

- メンバーとタスクの切り分け
  ここでは、タスクに関わる個人の性質が要求されているわけではないので、メンバーとタスクの関係は動作主体が1人である場合は動作を区別する必要がない。しかし、動作主体が複数でかつ協調する場合には動作主体と作業を組み合わせたスケジュールが必要である。そこで、タスクと動作主体を別の構造としてプロセスを起動するために、メンバーのプロセスへの割り当てを基本メカニズムとして定義する。

- 協調動作の記述
  デザインレビューの場合、チームでの仕事と個人の仕事がある。各人がレビュー作業を行なうわけであるが、他のメンバーとの意見交換をしながら作業をすすめる。そこで、個人からチームへのメッセージといった事象を記述する必要がある。また、レビュー作業の結果はこのような個人の作業を取りまとめたものである。すなわち、個々に作業を行なってその結果をまとめるタスクが必要である。

- 情報の伝達手段の記述
  入出力は情報の伝達手段が関わる。電子メール、口頭、手渡しの伝達手段がある。データ授受の出入口に適用する。

3.2 記述の方針

前記の分析結果から、つぎのような方針で記述を行なうことにする。

- 関数型の言語の基本メカニズムは関数適用だから、関数名（作業名）とデータ（入力）を与えてそれらの起動をマージさせる。関数適用はプロセスの起動になる。ここで関数名は個人のタスクとチームのタスクに相当するものがあると考えられる。

- 各プロセスは関数名とデータの組のリストとする。個人のタスクの場合は割り当てられた個々のメンバー毎に起動されるものと考える。

- 関数の組み立ての基本は、データと機能のスコープの決定である。すなわち、どのデータを機能のどの段階で扱うことが有効であるかを考察する必要がある。

そしてつぎのような視点で記述の作業を行なう。

- 各プロセスの要求を分析しデータやタスクを抽出する。

- プロセス間関連の関数を共通に定義する。このために基本データ型を定義する。
  - プロセスの起動
  - メンバーの割り当て
  - 伝達手段

- 各プロセス単位でタスクの集合を形成する。プロセスのタスクを関数適用により構成する。

- 各プロセスのデータを分析する

- プロセス間のデータの関連を定義する。型の同義を考察する。

- 各プロセスの作業を定義する。

3.3 インタラクションの定式化

各プロセスの定義については付録にその一部を記載する。Schedule_and_assign_tasks プロセスでは、タスクと要員の時間的順序を考慮した project_plans を作成し、各プロセスを起動している。

さて、本項では、各プロセス間のインタラクションに関する基本メカニズムについて例を示しながら説明する。

- スケジューリング、タスクのインターリーブ（並列動作）

- プロセスの起動（プロセス間の通信）

- メンバーのプロセスへの割り当て

- 協調動作

3.3.1 スケジューリング

|| plan は1つのプロセスの担当者と仕事の組である。
|| project_plans はこれらのプロセス間の関係をスケジューリングしたものであるので、plan のリストのリストである。
並列に処理されるプロセスは1つのリストとする。時間的順序はリストの並びで表現する。例えば

```
[[Plan A B],[Plan C D,Plan E F],..]::project_plans
B=[[Work B1 B2],[Work B3 B4,Work B5 B6],..]::works
```

となる。

```
project_plans == [[plan]]
```

```
plan ::= Plan participants works
works == [[work]]
work ::= Work task input
task ::= Task taskname | Teamtask taskname

participants == [member]
member ::= Memb name role [process]
name == string
role ::= P_manager|D_engineer|Qa_engineer|Other
```

||Team にメンバーを登録し、スケジュール時にその属性を更新する。

|| team を結成する場合は、上記のメンバーを適当に組み合わせる。例えば

```
Team = [Memb B D_engineer [Modify_design ,Review],
  Memb C Qa_engineer [Review],Memb D Other [Review],
  Memb E Other [Review,Modify_code]]::participants
```

のようになる。

||Team に登録されたメンバーから指定プロセスに関わるメンバーを選出する。

```
get_member :: procname → participants
get_member procname = get_team_member Team procname
get_team_member :: team → procname → participants
get_member team procname =
   [m | m ← team ; member procname get_process m]
```

|| 上記のような work のリストを適当にマージする。

```
planning :: [[*]] → [**]
planning  xlist = xlist          if length xlist = 1
                = permutation xlist   otherwise
```

3.3.2 プロセスの起動

|| 要求における入出力を分析し、プロセスのネットワークを構成する。

||Modify_desgin 等が各プロセス名を表す。右辺に現れるプロセスに束縛された値を順次処理してその結果を左辺のプロセスに束縛する。

```
Requirement_change
Team
```

はグローバルに設定し、随時参照する。

|| 入出力関係をまとめるとつぎのようになる。ここで ψ は、各プロセスで起動されるタスクを表す。この中でファイルの入出力を扱う。

```
Schedule_and_assign_tasks =
  ψ Requirement_change  where  read PPF
                               write PPF
Monitor_progress =
    ψ Review_design Modify_design Test_unit
      Modify_code Modify_unit_test_Package
      Modify_test_plans Schedule_and_assign_tasks
     where   read PPF
             write PPF
Review_design = ψ Modify_design  where write SDDF
Modify_design = ψ Review_design  where read SDDF
Test_unit = ψ          where read SDF
                             read TPaF
                             write THF
Modify_code = ψ Modify_design Test_unit
          where read SDF
Modify_unit_test_Package =
  ψ Modify_design Test_unit     where read SDF
                                      read TPIF
                                      read TPaF
                                      write TPaF
Modify_test_plans = ψ       where read TPIF
```

|| このようなネットワークによって各プロセスが出力するデータを他のプロセスが受け取ってタスクを行なう。

|| つぎに ψ の設計を行なう。プロセスのタスクの決定と入力の整理、タスクの起動の枠組の決定を行なう。

|| プロセスをある入力で起動する。結果を各プロセスのストリームに出力する。

```
process_wake :: participants → works → [output]
process_wake part works =
     [task_assign  part planning works]]
```

3.3.3 メンバーのプロセスへの割り当て

|| 仕事の参加者のタスクを生成する。仕事には個人とチームとしての仕事があるので、これを分配することによって作業を発生させる。

```
task_assign :: participants → works → works
task_assign participants works =
   [indiv_work participants wk | wk ← works]
                    if length participants = 1
   groupe_work participants works      otherwise

indiv_work :: participants → work
indiv_work part work = [] if get_input work = []
 apply get_task work get_input work    otherwise

groupe_work :: participants → works
groupe_work participants works =
       [team_task? participants wk| wk ← works]

team_task? :: participants → work → output
team_task? participants Work (Task task) [] = []
team_task? participants Work (Task task) input =
          [apply task input | p ← participants]
team_task? participants Work (Teamtask task) input
        = apply task input
```

3.3.4 協調動作

|| レビュー作業は、他のメンバーの意見を考慮したり、自己の意見を述べながら作業をすすめる。そしてある段階で結果を決定する必要がある。

|| この協調作業における個人のタスクは、他のメンバーの意見を聞いて自己の結論を出すことと、他のメンバーに意見を述

べることの2つである。

```
listen_to_review :: Set design [idea] → idea
comment_for_review :: Set design [idea] → idea
|| チームとしてレビュー結果を決定する。
decide :: [idea] → idea
decide [] = []
decide ideas = result_decide ideas
                 if length ideas > decision_number
               []                otherwise
result_decide  :: [idea] → idea
result_decide ideas = Result idea
|| Review_design プロセスにおけるレビュー作業はこうした
個人のタスクとチームとしてのタスクの結果を生成しながら進
められる。
ordinary_review ::
        modified_design → [idea] → reviewd_design
ordinary_review m_design ideas =
  get_idea (hd result)      if length result = 1
  ordinary_review m_deesign Ideas      otherwise
 where
  Teamreview =[[[Work (Task comment_for_review)
                              Set [m_design,ideas],
Work (Task listen_about_review) Set [m_design,ideas]],
  Work (Teamtask decide) ideas]]
  Ideas = task_assign (get_member Review_design)
                      planning Teamreview
  result = results? Ideas
|| results? は Ideas のリストの中に Result idea がある
かどうかを調べて、あればその結果を、なければ Ideas を返
す。
```

その他に伝達手段の記述について述べる。情報の伝達手段は、e_mail、verbal、hand_carried がある。どの場合もそれぞれの情報を伝達手段によって置き換えて相手のプロセスに送る。起動されたプロセスは情報を変換して利用する。

```
communication ::=
 Mail e_mail | Verb verbal | Hand hand_carried
recieve_notification::communication→notification
recieve_notification Mail notification=notification
...
recieve_notification [] = []
```

4 LOTOS によるプロセス記述との比較

LOTOS では、システムとやり取りする外部からの観測可能なイベント間の時間的順序を規定することによって仕様を記述する。キーワードはつぎのとおりである。

- プロセス
  環境における他のプロセスと通信を行なう抽象的な実体である。通信を行なうプロセスの作用点をゲートと呼び、ここでデータのやり取りを行なう。
- アクション
  インタラクションの基本単位であり、ゲートとデータの組からなる。アクションはアトミックであり、かつ同期的である。
- データ定義
  抽象データ型によるデータ定義を行なう。構造的な仕様を書くための枠組である。

LOTOS におけるプロセスの記述はつぎのように行なう。

- いくつかの階層で表現
- 上位においては各プロセスがどのゲートを介して関連するかを記述する。
- アクションの系列を記述する。ゲートと変数、または条件式（あるゲートにおけるある値の関係が成立すれば、情報をやり取りする。）によって構成する。

LOTOS でいうところのプロセスとアクションは Miranda における関数定義と関数名に相当する。Miranda では、関数定義によってある種のアクション系列をブラックボックス化することができる。

LOTOS では細かい単位の系列を記述することによって相互作用が定義される。Miranda の場合は、相互作用を定式化することによって、抽象化された枠組を提供し、再利用しやすい構造を与えている。

本例題で要求されているプロセスの記述はまだ大まかなものであり、実プロセスとの関連からこの記述をブレイクダウンしていく必要があると思われる。その意味では、関数はブレイクダウンしやすいと考える。

5 考察

- 抽象化・具体化による仕様化プロセス

Miranda のような関数型言語による記述においては、関数抽象と関数適用という単純なメカニズムによって柔軟度が高い構成が可能であると思われる。すなわち、関数をうまく構成すると、その部分的な機能を利用することができる。これは関数を小さな単位で記述し、それを組み上げることでさまざまな機能が実現できることを意味する。しかし、一方で組み上げられた関数は、状況の変化に対していつでも柔軟に対応できるわけではないし、構成しやすい関数を定義することも難しい。実際に本例題においてもデータの同義や関数を汎用化するために、データの拡張といった作業が頻繁に生じている。

そこで、プロセス記述に限らず、仕様化のプロセス

において具体的な問題に特化された記述を抽象化したり、それを他の状況に適合させることによって具体化するといった、仕様の構造化が必要であると考え、現在研究中である[5]。ここで言うところの抽象化は、例えば、関数のカリー化や関数の定義域の拡張による汎用化といったことで、インタラクションの定式化はソフトウェアプロセスにおける協調メカニズムの1つの抽象化であると考える。

• プロセス記述における抽象化の意味

本稿の目的の1つはプロセス記述の再利用であるが、上記で述べた抽象化・具体化による仕様化プロセスにはつぎのように適用されると考える。

1. ある分析／設計技法のプロセスを記述する。
2. 具体的なプロジェクトのプロセスを記述する。上記の技法の記述に基づいて、具体化によるプロジェクトの記述の生成を行なう。必要な環境や事例を定義する。
3. シミュレーションによって、ノウハウや事例をさらに蓄積する。
4. 蓄積した知識を抽象化することによって、はじめの記述に指針として追加する。
5. 成長したプロセス記述を他の具体的なプロセスに適用する。

• インタラクションの定式化の利用について

プロセスの部品化や再利用を行なうためには、プロセス記述が別の状況に適用可能である必要がある。これまで述べたようにソフトウェアプロセスにおけるインタラクションを定式化することによって、これを再利用することが可能である。以下の作業と定義されたインタラクションを使って仕様を構築する。

- プロセス間のネットワークの整理
  - 入出力データを関係づける。
- タスクの定義
  - プロセスのタスクを抽出する。
  - プロセスを work のリストとして定義する。
  - work の入力の初期値を設定する。
- データ定義
  - プロセス毎に入力と出力を整理する。abstract data type で定義する。
  - 型の同義、機能との関連のための定義域の拡張を考察する。

ここで特に問題となるのが機能の抽象化に伴うデータの変更である。この変更のパターンを分類することによって各機能が変化する場合の知識が収集できると考える。

6 まとめ

関数型言語 Miranda によるソフトウェアプロセスの記述を、プロセスの再利用の観点から実験した。本稿では関数型言語のもつ抽象化能力の1つとして高階関数の利用によるインタラクションの定式化が行なえたと考える。さらに、このような抽象化を利用した仕様化プロセスを明確にしたいと考える。また、今回は抽象データ型や代数的型定義を利用していないので、これらの役割についても考察したい。

謝辞

本研究は、次世代産業基盤技術研究開発「新ソフトウェア構造化モデルの研究開発」の一環として情報処理振興事業協会が新エネルギー・産業技術総合開発機構からの委託をうけて実施したものである。また、有益なご助言をいただいた、東京工業大学佐伯元司助教授と IPA 古宮誠一氏に感謝致します。

参考文献

[1] 荒木：並列動作システムの Miranda による仕様記述，コンピュータソフトウェア，Vol.8,No.1, pp. 12-24, 1991

[2] R.Bird and P.Wadler:Introduction to Functional Programming , Prentice-Hall,1988

[3] M.Kellner : Software process modeling :A case study ,Technical report ,1990

[4] D.A.Turner : An Overview of Miranda , SIGPLAN ,Vol.21 , No.12 , pp.158-166 , 1986

[5] 松浦, 本位田：形式的仕様記述言語における抽象化について，第43回情報処理学会全国大会，1991

[6] R.Milner:Proposal for Standard ML,Conferance Record of 1984 ACM Symposium on LISP and Functional Programming,ACM,pp.184-197,1984

[7] Miranda System Manual ,Research Software Limited ,1989

[8] M.Saeki,T.Kaneko and M.Sakamoto : A Method of Software Process Modeling and Description using LOTOS ,Proc. of First International Conference on the Software Process, pp 90-104,1991

A 記述例

```
|| Schedule and Assign Tasks プロセスの定義
project_plans == [[plan]]
plan ::= Plan participants works
update_project_plans == project_plans

update_project_plans :: notification → output
update_project_plans notification =
        s_a_a_task_output (notification_to_process put_file_data PPF ''project_plan''
                (schedule_and_assign_tasks notification Team c_project_plan))
    where   c_project_plan = get_file_data PPF ''project_plan''
            notification_to_process :: project_plans → [output]
            notification_to_process project_plans =
            [process_wake get_participants pp get_works pp | pp ← planning project_plans]
            get_participant Plan p w = p
            get_works Plan p w = w
            schedule_and_assign_tasks :: notification → participants → project_plans
            schedule_and_assign_tasks notification participants = .....
s_a_a_task_output :: [output] → output
s_a_a_task_output outlist = []    if out = []
                          out   otherwise
            where   out = remove_complete outlist

|| Review_design プロセスの定義
Review_design = process_wake (get_member Review_design)
                [[Work (Task review_design) recieved_notification Modify_design]]
review_design :: [design] → [output]
review_design [] = []
review_design [design : rest] =  review_design_out outcome review design ++ review_design rest

review :: design → reviewed_design
review  In m_design  = ordinary_review m_design []
review  Out Approved m_design o_notif  = []
review  Out Minor m_design o_notif  = perfunctory_review m_design []
review  Out Major m_design o_notif  =  ordinary_review m_design []

perfunctory_review :: modified_design → reviewed_design
ordinary_review :: modified_design → reviewed_design

outcome :: reviewed_design  → reviewed_design
outcome Approved m_design o_notify = Approved (put_file_data SDDF ''modified_design'' m_design) o_notif
outcome Minor m_design o_notify  = feed_back Minor m_design o_notif
outcome Major m_design o_notify  = feed_back Major m_design o_notif

feed_back :: reviewed_design → reviewed_design
feed_back r_design = process_wake Design_review_team [[Work (Proc Modify_design) Hand r_design]]

review_design_out :: reviewed_design → output
review_design_out Approved m_design o_notif = Mail onotif
review_design_out Minor m_design o_notif = []
review_design_out Major m_design o_notif = []
```