Vol. 34 No. 11 情報処理学会論文誌 Nov. 1993

ソフトウェア協調開発プロセスのモデル化とそれに基づく開発支援システムの試作

飯田 元[†] 三村圭一[†]
井上克郎[†] 鳥居宏次[†,††]

本論文では，多人数によるソフトウェアの協調開発において，各開発者の行っている作業間の同期や調整に着目したプロセスモデルの提案を行う．さらに，提案したモデルに基づいて各作業の進捗状況の記録・表示を行うシステムを試作する．提案するモデルは，開発過程を並行して進められる複数の作業系列（タスク）の集合とし，タスクどうしの同期や調整は通信動作によって行う．各開発者は，それぞれがいくつかのタスクを担当する．このようなモデルを用いて協調開発プロセスを記述することで，どのようなタスクが並行して進められ，どのような同期や通信が必要かを明らかにすることができる．このモデルをもとに作成されたシステムは，プロジェクト管理者用と各開発者用の二つの部分に分けられる．開発者用のシステムは，それぞれのタスクについてメニューによる作業の誘導やツールの起動などを行い，また，各作業の開始・終了を逐次，管理者用システムに報告する．一方，管理者用システムでは，タスク間の通信を中継・管理し，報告されたデータをもとに各タスクの進捗状況や各開発者の作業履歴を表示する．このシステムを利用することにより，各作業の誘導や作業間の進行の同期・調整などが行える．さらに，プロジェクト全体の進捗状況を把握するために必要な情報が得られる．

Modeling Cooperative Development Process and Prototyping a Monitor/Navigation System

Hajimu Iida,[†] Keiichi Mimura,[†] Katsuro Inoue[†] and Koji Torii[†,††]

This paper describes a process model for a cooperative software development performed by a group of developers, and also we discuss a prototype of a project monitoring system based on this model. The model proposed here is based on concurrent process one, and it is composed of set of tasks associated with communication primitives among the tasks. A task is defined as a sequence of primitive activities. Using this process model, a project monitoring system named "Hakoniwa" (which means diorama in Japanese) has been implemented. Hakoniwa system is composed of two parts—manager's system and developers' system. Developers' part navigates the developer's activities by showing possible succeeding activities on menu. By the menu-item selection, devolopment tools are executed automatically. The sequences of activities performed by each developer are recorded by the manager's system. The manager can observe the cooperative development process by using Hakoniwa system.

1. はじめに

ソフトウェアの大規模化や人件費の増大につれて，1箇所での集中したソフトウェア開発が非常に困難になりつつある．その結果，地域分散開発などの支援に関する研究が盛んに行われるようになってきた[1]．本研究は，ソフトウェア開発プロセスモデルを基礎にした情報管理を用いて，複数人での協調開発プロジェクトの管理・支援を効果的に行おうとするものである．

一般に，ソフトウェア開発プロセスとは，企画・設計段階から保守段階までがどのような工程で行われるかといったことや，各工程でどのようなドキュメントを参照し，どのような作業を行い，どのような成果物を作成するかといった事項を定めたものをいう．近年，Osterweil によるプロセスプログラミング[2]などをはじめとして，ソフトウェアの開発の過程（プロセス）で生じる種々の作業や判断，また，その

† 大阪大学基礎工学部情報工学科
Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University
†† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Advanced Institute of Science and Technology, Nara

環境などを抽象化して記述する試みが数多くなされている[2),5),12),13),16),23)]．プロセスのモデル化によって，複雑な開発手順や開発方法論を正確に他人に伝える[19)]ことや，開発プロセスの評価・解析を系統的に行う[18)]ことが期待でき，作成したモデルを開発支援環境の基礎制御構造として用いる[15),22)]こともできる．

さらに，協調開発プロジェクトの管理において，各開発要員がいまどのような作業を行っておりどの工程まで進んでいるか，といった進捗状況の把握をする場合にもプロセスの記述を判断の基準として用いることができる．

つまり，複数人の開発プロセスを特徴づける要素をプロセスモデルとして形式化することによって，このような作業が分担して行われ，どのような制御が必要であるかを明らかにすることができる．具体的には，それぞれの作業の手順と互いの関係を明確に定め，それに従った開発を誘導すると同時にその進行状況を逐次記録することで，開発の効率と信頼性を向上できるものと考えられる．

一方，今日では多くのソフトウェア・メーカーが，標準開発手順と呼ばれるプロセスモデルの一種を用いている[4)]．これらの標準開発手順は，製品の品質向上と工程管理の効率化をおもな目的として用いられているが，そのほとんどは上流工程から下流工程への一方通行的な進行を仮定した「ウォーターフォール・モデル」に基づくものである．

しかし，実際のプロセスにおいては各工程のやり直しや上流工程へのバックトラッキングなどが存在するため，必ずしも実際の開発形態に対応しない場合が多く[3)]，したがって進捗状況も把握しにくい．多くの場合，管理者が報告書などをもとにして判断を行っている．さらに，多くの標準開発手順では多人数による作業であることをとくに意識して定義されていない．たとえば，複数人による開発過程を考えた場合には，互いの作業の同期や制御が必要になるが，これらは明示されずに一般に各員の責任において行われている．したがって，複数人による協調的な開発をより効果的に支援するためには，並行して行われる工程や，開発者間でやりとりされるメッセージなどを明確に表現できるような新しいプロセスモデルが必要であろう．

本論文では，ソフトウェア開発とその管理におけるプロセスモデルの活用法一とくに，複数人での協調開発を行う場合のプロセスモデルについて論じる．まず，協調開発を対象としたソフトウェアプロセスモデルを提案し，さらに，提案したモデルに基づいて試作した協調開発支援・管理システム「はこにわ」について述べる．

以降，2章では協調開発を対象としたプロセスモデルの提案を行い，3章ではこのプロセスモデルに基づく開発支援と管理の手法について論じる．さらに，4，5章では支援・管理システムプロトタイプ「はこにわ」の概要とその適用例を示し，6章で若干の議論を行う．

2. ソフトウェアプロセスのモデル化

2.1 複合ソフトウェアプロセスモデル

これまでに，さまざまなソフトウェアプロセスモデルが提案されている．しかし，現実のプロセスは非常に複雑でさまざまな要素から成り立っているため，一つの平板なモデルですべての要素を適切に表現することは不可能である．そのため，提案されるモデルの多くが非常に高機能かつ複雑なものとなっている．しかし，文書化や形式化の観点から見た場合，複雑なモデルはそれ自体が理解しにくく，記述内容の評価も困難となる．

たとえば，HFSP[13)]のように，各工程をプロシージャや関数とみなし，それに対するパラメータとしてプロダクトを表現するモデルが数多く用いられている．このような場合，プロダクト間の全体的な生成関係を知るためには，プログラムにおけるデータフローの追跡と同様の作業が必要である．また，わかるのは依存関係のみで，プロダクトの親子関係や物理的な構成（ディレクトリの配置など）は直接定義できない．一方，Marvel[12)]のようにプロダクト間の関係規則を主体としたモデルでは，このような情報を容易に定義できる一方で，プロセスのもつ階層構造などが直接得られず，全体の作業構成が不明確となる．

これらの記述方法はプロセスのある特定の性質に主眼をおいたもので，作業の流れやプロダクト間の関係といった，特定の視点からプロセスを眺めるうえでは必要かつ十分なものであろう．しかし，モデルの記述能力を高めようとしてこれらを直接統合することは，プロセスの流れの指定のための記法やプロダクトの関係の指定のための記法といったさまざまな表現方法の混合につながり，逆に全体として見にくく理解しづらい記述が得られてしまうこととなる．

そこで，ソフトウェアプロセスのもつさまざまな要素を個別に単純なモデルとして実現し，次にそれらの

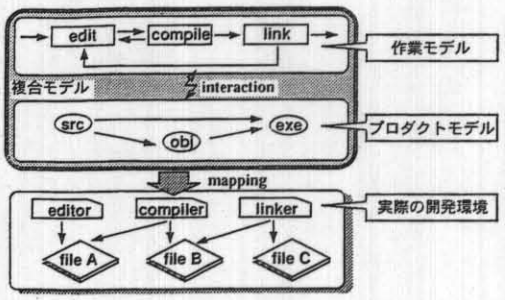


図 1 複合モデルの概念図
Fig. 1 Overview of composite software process model.

相互関係を定めたモデル（**複合モデル**と呼ぶ）を構成することが考えられる．たとえば，**図1**のように**作業モデル**と**プロダクトモデル**，およびその関係をそれぞれ定義し，必要に応じていずれかのモデルを選択して用いる．作業モデルは，開発で行われる作業間の順序関係を定義するためのもので，プロダクトモデルは，開発中に作成されるプロダクト間の関係を定義するためのものである．このように二つのモデルを分離することにより，作業手順の系統的な分類とプロダクトの論理的・物理的構造をそれぞれ明確に表現することができる．

本論文ではこの図1の例の複合モデルに基づいて，作業モデルのみを議論する．とくに多人数による開発の特徴を反映した作業モデルを提案する．

2.2 作業モデル

ここで提案する作業モデルは，基本的に Hoare の CSP (Communicating Sequential Processes)[6]等と同様に，並列に実行される複数の「タスク」によって構成される．

実際の開発プロセスは多くの工程から構成されている．いくつかの工程は複数の人間によって並列に行われる．また，開発者は互いに連絡をとりながら進行の調整や他の工程の制御を行う．このとき，開発者を主体にして工程の進行のモデルを考えると，一人の開発者が互いにあまり関係のない複数の作業を行うこともあるため，モデルが複雑になる．そこで，まず各工程を主体にモデルを構築し，開発者は単にいくつかの工程の担当者として捉える．

工程の捉え方の単位には，「ファイルを編集する」といった小規模のものから「システムの仕様を変更する」といった大規模かつ多人数で行うものまである．ここでは，プロセスの最小の構成単位（ツールの実行や人間による判断など）を「基本作業」(activity)として定義する．そして，並行した動作を伴わず，順次逐次的に行うことのできる基本作業の系列を「タスク」と呼ぶ．また，タスクの集合を一つのソフトウェア開発過程全体とする．各タスクは基本的に並列に実行される．このとき，タスク間の同期をとったり他のタスクを制御したりするために，基本作業の一種として同期/非同期の通信動作（文字列の送受信）をタスクにもたせることができる．各タスクに送られたメッセージは受信用ポート（キュー）に書き込まれ，メッセージを送信する際には送り先のタスクとその受信ポートを指定する．一つのタスクは複数の受信ポートをもつことができる．基本的な通信動作を**表1**に示す．

さらに，タスクの開始，終了の記述を簡略化するために，**表2**に示すようなタスク制御用オペレーションが用意されている．これらは基本通信動作を用いて実現されており，すべてのタスクに制御通信専用のポートが定義されている．

タスク内部の動作はこれらの通信動作を含めた基本作業の系列として定義する．ここでは系列は正規表現で表す．正規表現で表現できないような系列は，もっと小さな系列に分解してそれをタスクとし，タスクの集合とタスク間の通信によって表す☆．各開発者は複数のタスクを担当し，定義されている系列に従って基本作業を実行することで，各タスクを進行させていく．

「作業モデル」におけるタスク間の関係をより直観的に表現するために，**図2**に示すような図式表現を用

表 1 基本通信操作
Table 1 Message transfer primitives.

操作	引数	動作	戻り値
send	task,port	文字列を送信する(非同期)	——
recv	port	ポートから文字列を読みとる(同期)	文字列
peek	port	ポート内のメッセージの有無を調べる(非同期)	論理値

表 2 タスク制御用通信操作
Table 2 Task control primitives.

操作	引数	動作
start	task	他のタスクに開始要請を送る
wait	task	他のタスクの終了を待つ
exit	task	自タスクの終了を通知する

☆ 系列の言語的なクラスの妥当性については5章で議論する．

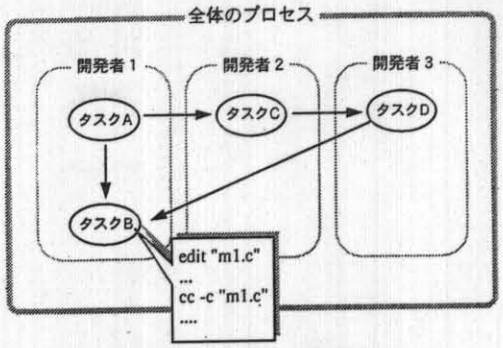


図2 作業モデルの概念図
Fig. 2 Overview of activity model.

いる．楕円がそれぞれタスクを表し，矢印がタスク間でやりとりされるメッセージを示している．メッセージの種類は矢印の横に自然語で記述する．

タスクのクラス化

タスクには，たとえば「モジュールAの変更」，「モジュールBの変更」といったように，入力や出力になるプロダクトは異なるが行う作業の手順は共通であるものが多い．この共通の手順をタスククラス（たとえば「モジュールの変更」）として定義し，タスクはタスククラスのインスタンスとする．モジュール名などといった各インスタンスに固有な値はインスタンス変数として定義する．

タスクの実働化

上述のようにして定義されたタスクは，タスクの構造の評価やタスク間コミュニケーションの解析などに役立つほか，実際にタスクの構造に従ってプロセスを実行することが考えられる．

しかし，本モデルはタスクの見かけ上の振舞いのみに着目したものであるため，記述されたプロセスを実行するには以下のような情報がさらに必要である．

（1） 各タスクの実入出力プロダクト
（2） 基本作業と開発ツールとの対応
（3） ループの終了条件や，選択条件

（1），（2）については，複合プロセスモデルの要素モデルとして以下のような外部仕様をもつプロダクトモデルを導入することにより，これらの情報の系統的な指定が可能となる☆．

- プロダクトモデルはプロダクトの集合Oである
- Oの各要素 O_i について，属性の集合 A_i と操作の集合 M_i が定義されている．

☆ 4章で紹介するはこにわシステムでは，プロダクトサーバが未実装のため，管理者が実ファイル名やツール名を指定するようになっている．

このとき，タスク集合Tとプロダクト集合Oとの対応関係を定義することにより（1）の情報が定められる．その結果，タスクにおける基本作業と各プロダクトのもつ操作との対応が導かれ，（2）に関する情報が得られる．

（3）については，筆者らが文献10）で提案した手法に基づき，「制約条件」という概念を用いる．制約条件とは文法における生成規則を適用するために満たされるべき条件をいう．たとえば，同じシンボルXについて二つの生成規則

$X \rightarrow a$
$X \rightarrow b$

が存在した場合に，それぞれの規則を適用する際に満たされている条件（制約条件）を付加し，実行時には制約条件の満たされている規則のみが適用される．ループ（X^+）の場合も上記のような生成規則に展開され，同様に処理される．制約条件の記法等については実システムに依存するため，ここでは省略するが，プロダクトモデルにおいて定義されるプロダクトの属性（A_i）を参照することによって簡潔な記述が期待できる．制約条件に基づいた系列の実働化の詳細については文献10）を参照されたい．

3. プロセスモデルに基づく支援と管理

本章では，プロセスモデル（作業モデル）を用いることにより，どのような支援や管理が可能となるかについて論じる．

3.1 開発者に対する支援

個々の開発者に対しては，以下のような支援を行うことが可能となる．

（1） 手順の誘導

開発者が複雑な仕事（作業モデルにおけるタスク）をいくつも抱えていると，

- 現在自分はどれだけの仕事を抱えているのか
- それぞれの仕事について，次に何をすべきか
- 中断されたままになっている仕事は何か

といったことを把握するのが困難になる．逆に，ある程度まとまりをもった仕事ごとに，以上のような情報を提供することで開発者に対する支援を行うことが可能となる．さらに，単純な作業についてはその手順を自動化することも可能である．

（2） 協調支援

作業モデルに従ってプロセスを記述することにより，

- 各タスク間にどのような連絡が必要か
- タスク間の連絡はどのようなタイミングで行われるべきか

といったことが明確になり，単純な連絡については自動化が可能となる．

3.2 管理者に対する支援

一方，プロジェクトの管理者には以下のような利点をもたらす．

- タスクの複雑度とプログラマの能力に応じて，タスクを適切に分散配置する際の目安になる．
- 分散したサイトで開発を行う場合に，サイト間連絡を極力減らす際の目安になる．
- タスク定義はプロジェクトの進捗に対する道標（マイルストーン）設定の手がかりとなり，タスクに対する監視（モニタリング）を行うことで，作業モデルに基づいた進捗や各開発者の負荷の調査が行える．
- 長期間放置されているタスクや，デッドロック状態を起こしているタスクの発見を容易にする．

本研究では，これらの支援のうち，開発者に対する支援と管理者に対する支援の一部（タスクモニタリング）を実現するシステムを実際に試作・試用した．次章ではこのシステムについて述べる．

4. 協調開発支援システムの試作

4.1 はこにわシステムの概要

前章で述べた作業モデルをもとに協調開発の支援を行うシステムはこにわを試作した．システムの構成を図3に示す．はこにわは，タスクのモニタリングとタスク間のコミュニケーションの中断をするはこにわサーバと，個々の開発者が用いるための支援環境タスクマネージャから構成される．タスクマネージャは，管理者によって割り当てられたタスクの実行エンジン（タスクドライバ）で起動・制御する．タスクドライバはプロダクトモデルから得られたプロダクトサーバを通じてプロダクトの参照・変更を行う．

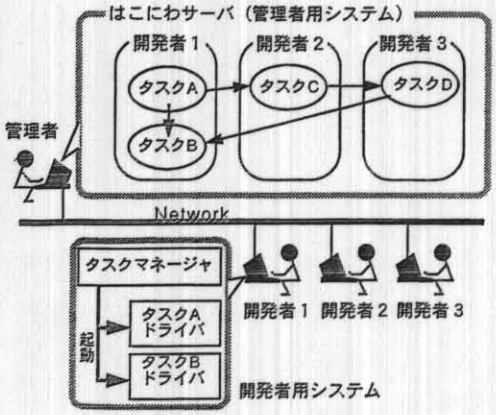


図3 はこにわ支援システムの概要
Fig. 3 "Hakoniwa" system architecture.

はこにわシステムは，作業モデルに基づくプロセス記述を入力として動作する．はこにわサーバには全タスク定義と，その割当て情報が入力として与えられる．各タスクマネージャには割り当てられたタスクのリストが与えられる．各タスクのドライバには共通の実行エンジンが用いられ，タスク内部の系列定義から得られる誘導プラン[10]が入力として与えられる．なお，現バージョンのはこにわではプロダクトサーバが未実装のため，入出力プロダクトやツール指定，制約条件などは管理者が事前に入力する必要がある．

このシステムでは主に次の三つの支援を行う．

（1） 作業の誘導

実際の開発では，複数のタスクを一人の開発者が担当する．そこで，まず管理者がタスクの割当を行い，その情報をもとに，各開発者のためのタスクドライバを生成する．各タスクドライバは，定義されている文法に基づく作業の進行を，メニューを通じて誘導する．メニューから選択された作業のうち開発ツールを用いるものは，自動的にツールの起動を行う．

（2） モニタリング

タスクドライバは，メニュー選択によって進行していく作業の履歴を記録する．この情報を参照することによって，開発者は現在進行中のタスクとその状態を把握することができる．すなわち，全開発者の作業情報ははこにわサーバで収集され，管理者がプロジェクト全体の状態を把握するのに役立てることができる．このとき，タスクごとの状態表示のほかに，開発者単位での全作業にわたる履歴の表示も行われる．

（3） コミュニケーション支援

タスク間のコミュニケーションは，すべてタスクマネージャ/ドライバとはこにわサーバを介して行われる．作業の開始要請や終了通知などといった，開発者間での単純なコミュニケーションは自動的かつ確実に行われる．

4.2 はこにわシステムの実現

複数の計算機上で行われる協調開発を想定して，タスク間通信の実現には UNIX のネットワーク機能（TCP/IP）を用いた．はこにわサーバはC言語で記述

されており，Sparc Station，Sun OS 4.1 上で動作する[7]．

タスクドライバには関数型の開発過程記述用言語 PDL[11),20)]のインタプリタを機能拡張して用いている．PDL インタプリタには，メニューによる作業の誘導やツールの自動起動などの機能が実装されており，今回の拡張でははこにわサーバとの通信機能を追加した．したがって，タスク実行に関する制約条件も PDL の式として記述する．

タスクマネージャもまた PDL プログラムとして記述されており，割り当てられた複数のタスクに対して，それぞれタスク定義から得られた誘導用の PDL プログラムを起動する．各タスクの起動はメニュー選択によって行われる．

タスク間通信機能の実現

タスク間の同期や他のタスクの制御は，通信機能を用いて実現する．通信機能はデータの送受信とタスクの実行状態の参照・制御の二つに分けられる．データの送受信には次のような事項がある．

- 送信できるデータは文字列型とする．
- 各タスクには受信用のキューを複数定義できる．
- データ送信には宛先のタスクとキューを指定する．
- 受信キューに対しては，データの読み取りと，有無の確認の2種類の動作が行える．

この機能で短いメイルなどのやりとりができる．また，複数の受信キューに対して個別にデータの有無を調べることができるので，イベントとしてタスク間の同期や他のタスクの制御に用いることもできる．

さらに，上記の通信機能を用いて，次のようなタスク制御機能を実現している．

- 他のタスクの開始要請の送信
- 自タスクの終了通知の送信
- 他のタスクの実行状態の確認

これらの通信は，はこにわサーバが中継・管理を行うサーバ・クライアント方式で実現している．

タスク進捗状況の表示機能

実行された基本作業の系列を時間順に表示するだけでは，タスクがどこまで進んでいるのかがわかりにくく，進捗情報としては不十分である．作業モデルでは，タスクの作業系列は正規表現で表されるので，はこにわシステムはその構造を表現する木を用いて，今まで何を行ってきたのか，現在何を行っているのかを表示する（図4）．

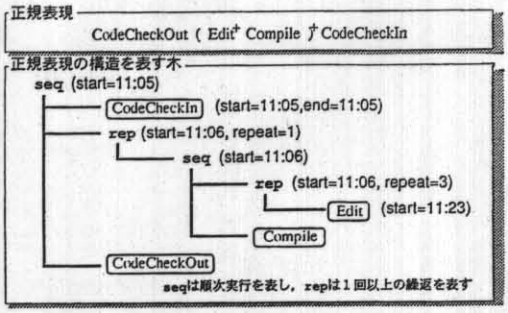


図4 正規表現の構造に基づいたタスクの表示
Fig. 4 Regular expression and its tree display for task.

図4は，最初にプログラムコードを取り出し，エディットとコンパイルを繰り返した後，コードを格納するというタスクの構造と，その履歴の表示例を表している．正規表現の構造木は複合設計やジャクソン法で用いられているものと同様で，逐次や繰り返しの部分に中間ノードを用いて階層的に表したものである．図4では **seq** が逐次を表し，**rep** が1回以上の繰り返しを表している．はこにわシステムは構造木の各ノードに対してはその開始時刻と終了時刻，さらに繰り返しの部分についてはその回数を記録し，表示している．この例では，Edit は 22：23 に開始されていて，これが3回目の繰り返しであることがわかる．このような表示を行うため，はこにわサーバは内部に各タスクの状態遷移図をもっており，常にタスクの実際の状態をトレースしている．

5. モデルの記述と実行の例

ここでは例として Kellner らによって提案されているプロセスモデリングのための例題[14)]を対象に，実際にモデルの記述を行った．この問題は，ソフトウェア開発過程に対するさまざまなモデル化の手法を比較・検討する手段を与える目的で作成されており，すでにいくつかの解答例が報告されている[17),23)]．

この問題全体は，あるソフトウェアシステムの一つのモジュールに対して変更を加える作業を規定しており，八つのサブステップに分けられている．各ステップにはそれぞれさまざまな開始・終了条件などが付加されている．また，ここで行う変更は比較的局所的で，あるモジュールを変更しても他のモジュールには影響を与えない．したがって，別のモジュールについて新たに変更作業を行ったり，関連するモジュールの整合性を調べるという作業は必要ない．プロジェクト

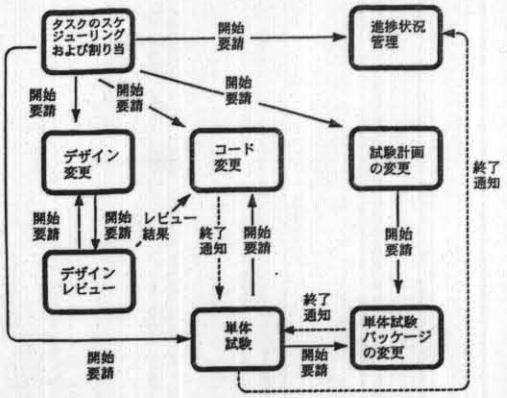


図 5 例題の作業モデルによる表現
Fig. 5 Activity model image for the example problem.

マネージャがスケジューリングし，仕事を割り当てることでこのプロセスは開始され，新しいコードがテストに合格すると作業全体が終了する．作業モデルによるこの問題の概念図を図5に示す．

この図の中で，**デザイン変更**，**コード変更**などがタスクである．これらのタスクは並行に行われる．また，『**デザインレビュー**が終了するまでに，**コード変更**は終了できない』，『**コード変更**と**単体試験パッケージの変更**が終了するまで，**単体試験**は開始することができない』といった．タスク間の進行に関する制限が存在する．この制限は，タスク間の通信によって実現する．図5では，見やすさのために終了通知メッセージを点線で表している．

これらのタスクの定義の記述例を図6に示す．たとえばタスク ScheduleAndAssignTasks では，最初に並行して行われる五つのタスクに対して開始要請を送信する．また，ModifyDesign では実際の変更作業(〈modify design〉)を繰り返した(＋記号で指定)後，ReviewDesign に開始要請を行う．これらの記述は，タスクのクラス定義を行ったもので，さらにモジュール名などのインスタンス変数を指定することで特定のプロジェクトに対応した記述となる．特定化された記述ははこにわサーバへの入力データとなり，さらに各開発者用のタスクマネージャ・タスクドライバ（ここでは PDL のプログラム）が得られる*.

得られたシステムをもとに，簡単な運用実験を行ってみた．タスクマネージャ／ドライバの起動によって，図7のようなメニューが開かれ，手順の誘導を行う．

* ここではプロダクトサーバを用いずに直接ファイルを操作するようなシステムを試作した．

```
ScheduleAndAssignTasks =
    <start MonitorProgress> <start ModifDesgin> <start ModifyCode>
    <start ModifyTestPlan> <start TestUnit>;
ModifyDesign =
    <modify design> + <start ReviewDesign>;
ReviewDesign =
    <review design>
    ( <start ModifyDesign> | <send "ModifyCode" "ReviewOK"> );
ModifyCode =
    ( ( <edit>+ <compile> )+ <peek "ReviewOK"> )+
    <recv "ReviewOK">;
ModifyTestPlan =
    <edit>+ <start ModifyUnitTestPackage>;
ModifyUnitTestPackage =
    <edit unitTestPackage>+;
TestUnit=
    ( <wait ModifyCode> <wait ModifyUnitTestPackage> <test>
      [ ( <start ModifyCode> | <start ModifyUnitTestPackage> |
          <start ModifyCode> <start ModifyUnitTestPackage> ) ] )+;
MonitorProgress = <wait TestUnit>;
```

図 6 作業モデルの記述例
Fig. 6 Activity model description for the example problem.

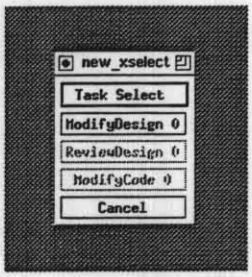


図 7 タスクマネージャによる誘導メニュー
Fig. 7 Menu displayed by the task manager.

図 8 はこにわサーバによる出力
Fig. 8 Output displayed by Hakoniwa server.

さらに，はこにわサーバではこれらの作業のモニタリングにより，図8のように情報を出力する．

6. 議　　論

本章では，作業モデルの適切さとその応用性，はこにわシステムの評価などについて議論する．

6.1 タスクの言語クラス

作業モデルでは，基本作業の系列として正規表現を用いた制約を行っている．しかし，系列のもつ構造がある程度複雑化すると，正規表現では表現することが不可能になる．このような場合，並列なサブタスクとして分割するか，タスク内部の表現に，より強力な言語クラス（たとえば文脈自由文法）を用いることが考

えられる．たとえば，プロトタイプ開発などのように差分的（incremental）な開発過程は文脈自由文法で再帰的に定義することで簡潔に表現できる[9),10)]が，正規表現では簡潔に表しにくい．また，有限オートマトンでは単純に表現できる系列が，正規表現に直すと複雑な表現となる場合もある．

しかし，正規表現を用いると，後で述べるようなデッドロックの検出などの静的な特性の解析を行いやすい．また，はこにわシステムでは，正規表現の木構造を用いてタスクの進捗データを表示しているが，文脈自由文法のように再帰構造を許すと木の深さが不定となるため，木の中の位置で進捗を判断することが困難になり，表示もわかりにくくなる．

6.2 計測環境としてのはこにわシステムの有用性

進捗の度合を測るためのデータとして，はこにわシステムでは，

- 繰り返しの回数
- 各作業の所要時間（開始時刻と終了時刻）
- 現在実行中の作業

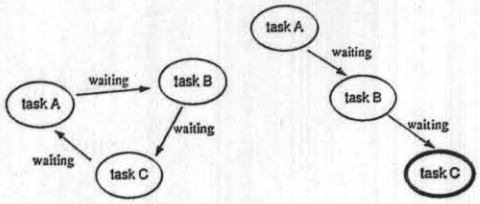
を収集できる．経験を積んだ管理者であれば，これらの情報から直観的に進捗を判断できるが，そうでない場合にはこれらの情報から何らかの定量的な指標を得ることが望ましい．しかし，単一のプロジェクトだけでこれらの値を収集しても，見積もりは困難である．

たとえば，ある作業についての現在の繰り返し回数がわかったとしても，最終的に何回の繰り返しが行われるのか事前にわからない以上，直接それを進捗の目安として用いることはできない☆．このような場合，過去のデータから平均的な繰り返し回数がわかっていれば，ある程度の予測を立てることができる．また，タスクのクラスごとに過去の平均的な所要時間（＝コスト）を求めるといったことも可能である．

このように統計的な予測を考える場合，プロジェクトの履歴を経験として蓄積することが重要となるが，はこにわシステムではデータの収集が自動的に行われるため収集に要するコストが低く，また，自己申告に基づくようなデータに比べて信頼性が高い．また，作業モデルに基づいたデータの分類・評価が可能であるという利点も生じる．

6.3 作業モデルにおけるデッドロック

すでに述べたように，ソフトウェアプロセスを形式的に定義することには，プロセスを系統的に解析することが可能になるという利点がある．たとえば，ソフトウェアプロセスにおける「手詰り」の状態を作業モデルにおけるデッドロックと捉えることができる．プロセス実行時におけるデッドロックの存在の多くは，はこにわサーバからの情報をもとに知ることができるが，モデルの記述からあらかじめデッドロックの起こる可能性を調べられることが望ましい．並行プロセス間のデッドロック問題はオペレーティング・システムなどの分野で従来から数多く研究されており，その手法を応用することができるであろう．



(a) Circular waiting　(b) Linear waiting

図 9 Circular waiting と Linear waiting
Fig. 9 Circular waiting and linear waiting.

ここでは，デッドロックを「送られてこないメッセージを待ち続けることにより，ある時点 t 以降，タスクの実行が永久に停止してしまうこと」と定義し，検出方法の例を示す☆．たとえば，メッセージ待ちの依存グラフを考えることによって，デッドロックを以下のように分類できる（図 9）．

- Linear waiting—経路が始点をもつ場合（ただし，始点は複数存在しても構わない）
- Circular waiting—経路が始点をもたない（有向閉路のみで構成される）場合

記述に対する静的な検出

これらのデッドロックの存在を静的に証明するために次のような十分条件，必要条件が考えられる．

- Linear waiting の十分条件：タスク T_i，T_j において，T_i の作業系列中にメッセージ m の送信が含まれていないにも関わらず，T_j が m の受信動作をもつ
- Circular waitingの必要条件：タスク間メッセージによる有向閉路が存在する

しかし，これらの条件は実際には有向閉路をもつ例が数多く存在するなど実用的でない場合も多い．そこ

☆ とくに繰り返しが多い，まったく実行されていない，といった異常な事態は検知することができる．

☆ プロダクトモデルは作業モデルとは別に存在するので，プロダクトへの排他的操作などによるデッドロックはここでは考慮しない．つまり，実行されないタスクの問題や，メッセージに対する circular waiting などを考える．

で，問題を簡単にするために，開始要請と終了通知のメッセージのみに着目した場合，以下のような十分条件がさらに存在する．

- Linear waiting：開始要請のみによる有向グラフを考えた場合に，最初に起動されているタスクから到達不可能なタスクが存在する
- Circular waiting：終了通知待ちの依存による有向グラフを考えた場合に，有向閉路が存在する

実行時の動的な検出

実行時に linear waiting が起きていないか検出するためには，その時点でのメッセージ待ち依存グラフの各始点のタスクがもつ履歴とその系列の定義を調べ，受け手のタスクが待っているメッセージが送られる可能性があるかどうかを判定する．

また，circular waiting が起きているかどうかは，その時点でメッセージ待ちによる有向閉路が存在するかどうかによって容易に判断できる．とくに，はこにわシステムの場合には中央のはこにわサーバによって進捗状況の管理とメッセージの中継が行われているので，検出は容易である．

7. おわりに

本研究では，多人数による協調開発の管理においてソフトウェアプロセスモデルを用いることの有用性を論じるとともに，プロセスモデルに基づいて管理を行うシステムの作成について述べた．

試作を行ったはこにわシステムでは，各開発者の行っている工程間の同期や調整と，各工程の進捗状況の記録・表示を行う．このシステムの利用により，手順の誘導や自動化などといった支援と同時に，プロジェクト全体の進捗状況を把握するうえで有益な情報が提供される．

さらに，これらの情報はソフトウェアプロセス自体の評価や開発者の作業量の見積もりなどを行う際にも有用であると思われる．

今後はタスクドライバの機能拡張（中断機能，複数タスクの非同期管理機能の強化）と，実際的な運用・評価を予定している．

また，提案した作業モデルではタスク間におけるプロダクトの受渡し関係等については対象外であったが，これらについてはすでに提案しているプロダクトモデル[8]に基づいたレポジトリを想定し，モデル間のインターフェースを定めた上で記述を行えるよう拡張する予定である．

参考文献

1) 青山幹雄：分散開発環境：新しい開発環境像を求めて，情報処理，Vol. 33, No. 1, pp. 2-13 (1992).
2) 安達久人，浜田雅樹：保守支援のための設計プロセス獲得システム，ソフトウェアシンポジウム '92 論文集，ソフトウェア技術者協会，長野，pp. E 11-18 (1992).
3) Boehm, B. W.: A Spiral Model of Software Development and Enhancement, *Computer*, May 1988, pp. 61-72 (1988).
4) Cusumano, M. A.: *Japan's Software Factories*, Oxford University Press (1991).
5) 櫨山淳雄，古宮誠一：プロジェクト管理のためのソフトウェアプロセスのモデル化について，第43回情報処理学会全国大会講演論文集（5），pp. 371-372 (1991).
6) Hoare, C. A. R.: *Communicating Sequential Proceses*, Prentice-Hall (1985).
7) ソフトウェアシンポジウム '92 ツール展示プログラム，ソフトウェア技術者協会，長野，p. 2 (1992).
8) Iida, H., Nishimura, Y., Inoue, K. and Torii, K.: Generating Software Development Environment from the Descriptions of Product Relations, *Proc. COMPSAC '91*, Tokyo, Japan, pp. 487-492 (1991).
9) Iida, H., Ogihara, T., Inoue, K. and Torii, K.: Generating a Menu-oriented Navigation System from Formal Descriptions of Software Development Activity Sequence, *Proc. 1st Int. Conf. Software Process*, Redondo Beach, CA, pp. 45-57 (1991).
10) 飯田，荻原，井上，鳥居：ソフトウェア開発作業系列の形式的定義と誘導システムの生成，情報処理学会論文誌，Vol. 34, No. 3, pp. 523-531 (1993).
11) Inoue, K., Ogihara, T., Kikuno, T. and Torii, K.: A Formal Adaptation Method for Process Descriptions, *Proc. 11th Int. Conf. on Software Engineering*, Pittsburg, PA, pp. 145-153 (1989).
12) Kaiser, G. E. and Feiler, P. H.: An Architecture for Intelligent Assistance in Software Development, *Proc. 9th Int. Conf. on Software Engineering*, Monterey, CA, pp. 180-188 (1987).
13) Katayama, T.: A Hierarchical and Functional Software Process Description and Its Enaction, *Proc. 11th Int. Conf. on Software Engineering*, Pittsburg, PA, pp. 343-352 (1989).
14) Kellner, M.: Software Process Modeling Example Problem, *Proc. 1st Int. Conf. on Software Process*, Redondo Beach, CA, pp. 176-

186 (1991).
15) Matumoto, Y. and Ajisaka, T.: A Data Modeling in the Software Project Database Kyoto DB, *Advances in the Software Science and Technology*, JSSST (eds.), Vol. 2, pp. 103-121 (1990).
16) Nakagawa, A. T. and Futatsugi, K.: Software Process à la Algebra: OBJ for OBJ, *Proc. 12th Int. Conf. on Software Engineering*, Nice, France, pp. 12-23 (1990).
17) 中山高史, 東野輝夫, 谷口健一: LOTOS によるソフトウェアプロセスの全体記述と開発者個人毎のプロセス記述の導出, 信学技法, Vol. SS 91-22, pp. 59-67 (1991).
18) 西村一彦, 本位田真一: 複合ビューポイントに基づく仕様化プロセスの定性的分析, 信学技法, Vol. SS 91-15, pp. 41-48 (1991).
19) 望月純夫, 山内 顕, 片山卓也: 人工衛星チェックアウト・システムの基本設計プロセスのプロセス・モデル HFSP による記述とその評価, 情報処理学会論文誌, Vol. 33, No. 5, pp. 691-706 (1992).
20) 荻原剛志, 井上克郎, 鳥居宏次: ソフトウェア開発を支援するツール起動自動制御システム, 信学論, Vol. J 72-D-I, No. 10, pp. 742-749 (1989).
21) Osterweil, L.: Software Processes Are Software Too, *Proc. 9th Int. Conf. on Software Engineering*, Monterey, CA, pp. 2-13 (1987).
22) 落水浩一郎: ソフトウェアプロセスモデルに基づくソフトウェア開発支援環境 Vela, 日本ソフトウェア科学会第 7 回大会論文集, pp. 205-208 (1990).
23) Saeki, M., Kaneko, T., Sakamoto, M.: A Method for Software Process Modeling and Description Using LOTOS, *Proc. 1st Int. Conf. on Software Process*, Redonodo Beach, CA, pp. 90-104 (1991).

(平成 4 年 6 月 26 日受付)
(平成 5 年 9 月 8 日採録)

飯田 元 (正会員)

昭和 63 年大阪大学基礎工学部情報工学科卒業. 平成 2 年同大学大学院博士前期課程修了. 同年同後期課程入学. 平成 3 年大阪大学基礎工学部情報工学科助手, 現在に至る. 工学修士. ソフトウェア開発プロセスおよび開発支援環境の研究に従事. 日本ソフトウェア科学会会員.

三村 圭一

平成 4 年大阪大学基礎工学部情報工学科卒業. 現在, 同大大学院博士前期課程に在学中. ソフトウェアプロセスのモニタリングに関する研究に従事.

井上 克郎 (正会員)

昭和 54 年大阪大学基礎工学部情報工学科卒業. 昭和 59 年同大大学院博士課程修了. 同年同大学基礎工学部情報工学科助手. 現在助教授. 昭和 59〜61 年の間, ハワイ大学情報工学科助教授. ソフトウェア開発環境, ソフトウェアプロセス等の研究に従事. 工学博士. 電子情報通信学会, ACM, IEEE 各会員.

鳥居 宏次 (正会員)

昭和 37 年大阪大学工学部通信工学科卒業. 昭和 42 年同大大学院博士課程電子工学専攻修了. 同年電気試験所 (現電子技術総合研究所) 入所. 昭和 50 年ソフトウェア部言語処理研究室室長. 昭和 59 年大阪大学基礎工学部情報工学科教授. 平成 3 年奈良先端科学技術大学院大学情報科学研究科教授 (大阪大学併任) 図書館長, 現在に至る. 昭和 41 年度早稲田賞, 昭和 51 年度および平成 4 年度電子情報通信学会論文賞受賞. 工学博士. ソフトウェアメトリクスやフォーマルな開発プロセスなど, 定量的取り扱いを中心とするソフトウェア工学に興味を持つ. IEEE Transaction on Software Engineering および IEEE Software 誌などの編集委員. IEEE, ACM および電子情報通信学会各会員.