レジスタ付ペトリネットモデルで記述されたソフトウェアプロセスの実行支援

山口弘純　岡野浩三　東野輝夫　谷口健一
{h-yamagu, okano, higashino, taniguchi}@ics.es.osaka-u.ac.jp
大阪大学　基礎工学部　情報工学科
〒560　大阪府豊中市待兼山町 1-3

計算機を用いてソフトウェアプロセスの実行を支援する場合，技術者の作業内容とそれらの実行順序に加え，技術者間のデータ交換などの連絡作業の内容とそれらの実行順序を定める必要がある．しかし，これらの連絡作業を陽に記述するのは繁雑で誤りも多いため，我々は，レジスタ付ペトリネットモデルを用いてプロセス全体の要求仕様を連絡作業なしで記述し，その記述から各技術者における連絡作業を含む動作仕様を自動生成する方法を考案している．本稿では，考案した手法に基づき，技術者の動作仕様群を自動生成するシステムと，それらの動作仕様群を分散実行してプロセスの実行支援を行うシステム (実行支援系) の設計の概略について述べる．設計した実行支援系の実働化により，技術者の負担が軽減され，作業効率が上がることが期待できる．また，現実のソフトウェア開発を考慮した機能設計を行うことで，より実用的な開発支援を実現することを目指す．

Enaction of Software Process Description in a Petri Net Model with Registers

Hirozumi Yamaguchi, Kozo Okano, Teruo Higashino and Kenichi Taniguchi
Department of Information and Computer Sciences, Osaka University
Machikaneyama 1-3, Toyonaka, Osaka 560, JAPAN

In order to enact a software process in a distributed environment, it is necessary to specify the engineers' work, communications between the engineers, and their temporal orders in the process. However, since it is so complicated to describe the communications, the designers may make mistakes in the process description. We have proposed a method to derive correct engineers' individual descriptions with communications from the whole description of a software process without communications, using a Petri net model with registers. In this paper, we have developed a derivation system according to the proposed method, and designed a support system which executes the derived individual descriptions in a distributed environment in parallel. It is expected that the efficiency of the engineers' work is improved since the engineers' burden becomes light using these systems. The goal of this study is to provide more practical support facilities for the software development.

1 はじめに

近年, ソフトウェアの大規模化や複雑化に伴い, グループによりソフトウェアの開発, 変更, 保守などを行う機会が増加している. そのようなソフトウェアの開発工程を一つの計算プロセスとしてとらえ, さまざまなアプローチでモデル化するソフトウェアプロセスに関する研究がさかんに行われている[1, 2, 3, 4, 5, 6, 7, 8, 12]. これらの研究は, 開発工程をプロセスとして明確にモデル化することにより, 曖昧さをなくし解析を容易にすることや計算機による実行支援を行うことなどを目的としている.

ソフトウェアプロセスを形式的に記述する方法として, プロセスの各ステップをツールの起動などの基本作業とみなし, これらの作業内容およびそれらの実行順序を指定することによりプロセスの記述を行う方法が広く行われている[1, 3] これらのモデルでは, プロセス記述に各技術者の作業の内容とその実行順序の指定, そして各技術者が指定通りに作業を実行するための技術者間の連絡作業を記述する必要がある. このうち, 技術者間の連絡作業の記述は各技術者でのデータ交換や作業の発生, 終了報告を考慮しなければならず, またそれらが必要となる頻度も高いため, それらの記述を陽に行うのは繁雑で誤りも多い.

我々の研究グループでは, 文献 [8, 12] で, 分散システムの仕様記述言語である LOTOS を用いて, ソフトウェアプロセス全体の要求仕様を技術者の作業内容とそれらの実行順序のみで記述し (これを全体記述と呼ぶ), その記述から各技術者ごとの動作仕様 (その技術者が行うべき作業と連絡作業, それらの実行順序が指定されたもの. これを個人プロセス記述と呼ぶ) を機械的に導出する手法を提案してきた. それらの手法により, プロセス記述者は連絡作業を陽に記述する必要がなくなり, プロセスを簡潔に記述することができ, かつ連絡作業の指定ミスを防ぐことができるようになった. また, 導出した個人プロセス記述では各技術者が行うべき作業が明確に記述され, 一般には複雑である技術者ごとの作業量評価も容易に行える. しかし, 文献 [8, 12] の方法では, データベースなどのリソースを特定の技術者のノード (計算機) に分散配置することができなかった (すべてのノードがデータベースを保持していると仮定) . またプロセスの動的修正も考慮されていなかった.

そこで, 本稿では, 我々が文献 [14] で提案した手法に基づき, レジスタ付ペトリネットモデルで記述されたソフトウェアプロセスの全体記述から, 各技術者ごとの個人プロセス記述を自動生成するシステムと, 生成した個人プロセス記述を複数の計算機上で分散実行支援するための実行支援システムの設計の概略を述べる. 提案した手法では, 入力として, プロセスの全体記述の他に, 各技術者のノードへのリソースの配置を指定することができる. したがって, 自動生成システムでは, 同じ全体記述に対し様々なリソースの配置指定を与えて, それに対する個人プロセス記述の導出を行うことができ, それらの比較評価を容易に行えるなどの特長を持つ. また, 実行支援システムでは, (1) プロセス情報の取得のためのプロセスの可視化機能, (2) プロセスの実行制御および技術者間の連絡作業の自動実行機能, (3) 作業の誘導およびそれに必要なツールの自動起動機能, (4) プロセスの実行中断 (状態退避) および回復機能によるプロセスの動的修正機能, などソフトウェア開発の支援に必要と思われる機能を実現することで, より実用的な支援システムを目指す.

以下, 2 章では, 個人プロセスを導出するためのアルゴリズムについて, 3 章では, システムの設計の概略についてそれぞれ述べる.

2 プロセス記述モデルと各技術者のプロセス記述の生成法

2.1 レジスタ付ペトリネットモデル

ペトリネット[9]は複雑な並列同期や選択を記述できる記述能力の高いモデルである. 我々は文献 [14] でペトリネットにゲートとレジスタを与えてデータを扱えるように拡張したレジスタ付ペトリネットモデルを提案している.

本稿では, 各技術者の作業はすべて計算機と技術者との間の入出力動作あるいは計算機内のファイルなどのリソースの更新操作として表す. したがって, ファイルのプレビューアや, エディタなどのツールを介した入出力動作が記述できるよう, 提案したモデルを若干拡張する.

拡張したレジスタ付ペトリネットモデル (Petri Net Model with Registers, 以下 PNR モデルと呼ぶ) は, 有限個のレジスタとゲートを持つ. このモデルでは, 従来のトークンによるトランジションの発火制御に加え, 各トランジションに与えられるガードと呼ばれる述語により発火制御を行うことができる. 各トランジションはそのガードの値が真でかつ発火に必要なトークンが揃ったときのみに発火可能であるとする. 各トランジションが発火すると, そのトランジションに与えられている入出力動作とレジスタ値更新式が実行される.

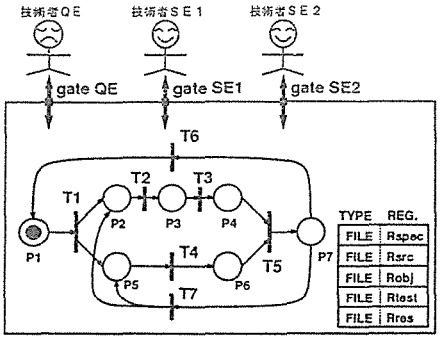
入出力動作は, 以下の形で表される.

$$g?x_1!E_1(\cdots)\ldots[x_1 = tool1(\cdots),\ldots]$$

g はゲート (技術者) 名, $?x_1$ は 入力変数 x_1 へのゲート g からの入力動作を表し, $!E_1(\ldots)$ は式 $E_1(\ldots)$ の値のゲート g への出力動作を表す (E_1 には (複数の) レジスタを引数として持つ関数を記述できる) . ここで, 既存のレジスタ R_1 をもとに, ツール $tool1$ で作成される新しいデータ x_1 を入力としたい場合は [ ] 内に $x_1 = tool1(R_1)$ のように記述する. 特別な動作として, いずれのゲートに対しても入出力動作を行わない内部動作 i も導入する.

レジスタ値更新式は以下の形で表される.

$$R_{h_1} \leftarrow f_{h_1}(\cdots),\ldots,R_{h_n} \leftarrow f_{h_n}(\cdots)$$



トランジション	ラベル		
	ガード	入力／出力動作	レジスタ値更新式
$T1$	TRUE	$QE?x_1[x_1 = \mathbf{edit}(R_{spec})]$	$R_{spec} \leftarrow x_1$
T2	TRUE	$SE_1!\mathbf{view}(R_{spec})?x_2[x_2 = \mathbf{edit}(R_{src})]$	$R_{src} \leftarrow x_2$
T3	TRUE	$SE_1!\mathbf{view}(R_{src})$	$R_{obj} \leftarrow \mathbf{compile}(R_{src})$
T4	TRUE	$SE_2?x_4[x_4 = \mathbf{edit}(R_{test})]$	$R_{test} \leftarrow x_4$
T5	TRUE	$QE!sizeof(R_{obj})$	$R_{res} \leftarrow \mathbf{exec}(R_{test}, R_{obj})$
T6	x_6 ==“accept”	$QE!\mathbf{view}(R_{res})?x_6$	
T7	x_7 ==“feedback”	$QE!\mathbf{view}(R_{res})?x_7$	

図 1: PNR モデルによるプロセス ModifyCode の 全体記述の例

各関数 f_{h_i} はレジスタと入力変数を引数に持つことができる. 記述されたすべての代入文は並列に実行される (すなわち, 並列に値が更新される) . 初期マーキングにおいては, 各レジスタは初期値関数により定められるデータ型の値を持つ.

2.2 プロセス全体の記述例

本節では, 3 人の技術者 (2 人のシステムエンジニア SE_1, SE_2 と 1 人の品質保証エンジニア QE) が, システムの変更を行う簡単なソフトウェア開発プロセス ModifyCode の全体記述を PNR モデルによりモデル化した例を挙げる.

プロセス ModifyCode は以下のような一連の作業を表すものとする.

1. QE がシステムの旧仕様書から新仕様書を作る
2. SE_1 が新仕様書をもとにソースコードを変更する
3. ソースコードからオブジェクトコードを生成する
4. 2., 3. と並行に, SE_2 が新仕様書をもとにテストデータを変更する
5. オブジェクトコードのテストを行う
6. QE がテスト結果により変更作業を終了するか, 2.～5. をやり直すかを決定する

図 1は, このプロセス全体を PNR モデルで記述した例である. 各技術者 QE, SE_1, SE_2 はそれぞれゲート QE, SE_1, SE_2 を用いるものとする.

まず, トランジション $T1$ が発火すると, ゲート QE において, 旧仕様書 (レジスタ R_{spec} の内容) から, 新仕様書が編集され, もとの R_{spec} に格納される. 次に, $T2$ では, ゲート SE_1 において, R_{spec} の内容が出力され, これをもとに, 旧ソースコード (レジスタ R_{src} の内容) から新ソースコードが編集され, もとの R_{src} に格納される. その後, $T3$ では SE_1 に新ソースコード (R_{src} の内容) が表示され, そのソースコードのコンパイルが行われる. 生成されたオブジェクトコードはレジスタ R_{obj} に格納される. また, $T2$, $T3$ とは並行に, $T4$ では, ゲート SE_2 にレジスタ R_{spec} の内容が出力される. これをもとに, 旧テストデータ (レジスタ R_{test} の内容) から新テストデータが編集され, もとの R_{test} に格納される. それらが終ると, $T5$ では, QE にオブジェクトコード (レジスタ R_{obj} の内容) のサイズが表示され, レジスタ R_{test} を用いた R_{obj} のテストが実行される. その結果は, レジスタ R_{res} に格納される. 最後に, テスト結果 R_{res} が QE に表示され, もし, テストに合格していれば $T6$ が発火して初期マーキングに戻り, そうでなければ $T7$ が発火し, $T2$ と $T4$ から同じ動作のやり直しとなる.

2.3 各技術者のプロセス記述の生成法の概略

本節では, 我々が文献 [14] で提案した手法にもとづき, 同じ PNR モデルで記述された各技術者のプロセス記述を導出するアルゴリズムの概略を述べる. 文献 [13] では, ソフトウェアプロセスの代表的例題である Kellner の例題[10]の全体を PNR モデルを用いて記述し, 各技術者のプロセス記述を導出し, その評価を行っている. 詳細はそれらの文献を参照のこと.

2.3.1 ゲートとレジスタの配置指定

導出アルゴリズムの入力として, プロセスの全体記述の他に, レジスタやゲートをどの技術者に配置するかの指定を与える (これを配置指定と呼ぶ). 提案した手法では, 簡単のため, 1つのゲートは1つの技術者のみに属するという制約を与える. 以下は配置指定の例である.

表 1: ゲートとレジスタの配置指定

技術者	QE	SE_1	SE_2
ゲート	QE	SE_1	SE_2
レジスタ	$R_{spec}, R_{test}, R_{res}$	R_{src}, R_{obj}	R_{src}

また技術者間の連絡作業は, すべてメッセージの送受信で行われるものとし, 任意の二人の技術者 eng_i, eng_j $(i \neq j)$ の 間にはそれらの連絡作業に用いる通信路が存在するものと仮定する. これを, 誤りのない無限容量の FIFO キュー $queue_{eng_i \to eng_j}$ と, その両端のゲート $G_{eng_i \to eng_j}$ により表し, 技術者 eng_i が $queue_{eng_i \to eng_j}$ にメッセージ M を書き込む時は $G_{eng_i \to eng_j}!M$, 技術者 eng_j が $queue_{eng_i \to eng_j}$ の先頭のメッセージ M の内容を入力変数 w に取り出す時は $G_{eng_i \to eng_j}?w$ と記述する.

2.3.2 各技術者のプロセス記述の導出

アルゴリズムは, プロセスの全体記述と, レジスタとゲートの配置指定を入力とし, 各技術者の個人プロセス記述を出力する. この際, 全体記述の PNR モデルのネットは活性安全な自由選択ネット[9]でなければならないなど, アルゴリズムの簡単化のためプロセスの全体記述と配置指定にいくつかの制約を与える. それらについては, 文献 [14] 参照.

与えられた制約を満たす全体記述と配置指定に対し, 以下の方法で個人プロセス記述を導出する.

[Step1] 全体記述の1つのトランジションに対し, 各技術者はそれぞれサブネットを得る. そのサブネットは, 通信路を介してメッセージをやりとりしながら (すなわち連絡作業を行って) そのトランジションと同じ作業を行う.

[Step2] 各技術者は, 全体記述と同形のネットを得て, そのネットの各トランジションと対応するサブネットを置き換える.

図2は, 図1の **ModifyCode** の全体仕様と, 表1の配置指定から, 上記のアルゴリズムにより導出した各技術者 QE, SE_1, SE_2 の個人プロセス記述である.

以後, 全体記述のトランジション T に対応する 技術者 eng のサブネットを T_{eng} で表す. また, サブネット T_{eng} 内で ID i を持つトランジションを $T_{eng}(i)$ で表すとする. 以下では, このプロセス **ModifyCode** を例にとり, 上記のアルゴリズムの各ステップについてやや詳細に説明する.

[Step1] ここでは, 全体記述のトランジション $T2$ を用いて, その $T2$ の作業を技術者間でどのように連絡作業を行って模倣するかを説明する.

まず, $T2$ の入出力動作 $SE_1 \,!\, \mathrm{view}(R_{spec}) \,?\, x_2[x_2 = \mathrm{edit}(R_{src})]$ は, 入出力ゲート SE_1 が割り当てられている技術者 SE_1 が行う (図2のトランジション $T2_{SE_1}(a)$ により実行される). この技術者は配置指定の制約 (2.3.1 節参照) より一意に定まる. これを $T2$ の責任者と呼ぶ.

次に, $T2$ のレジスタ値更新式 $R_{src} \leftarrow x_2$ は, 更新されるレジスタ R_{src} を保持する技術者 SE_1, SE_2 がそれぞれ独立に実行する $(T2_{SE_1}(b), T2_{SE_2}(b))$. ここで, 技術者 SE_2 はその実行に必要な x_2 の値 を知らないため, 責任者 SE_1 からその値をメッセージ M_9 として受け取る $(T2_{SE_1}(c), T2_{SE_2}(a))$. ここで, もし更新に必要な値を責任者が保持していない場合, それを保持している技術者から送ることにする. この場合, 入出力動作の終った責任者がそのきっかけとなるメッセージを送る. このように, レジスタ更新式の実行に必要でかつそれを実行する技術者が保持していないレジスタ値または入力変数値はそれを持っている技術者から受け取る. 更新が終った SE_2 は, 次の責任者 SE_1 に更新が終了したことを知らせるメッセージ M_{10} を送る $(T2_{SE_2}(c))$. SE_1 はそれを受け取り $(T2_{SE_1}(d))$, $T3$ の入出力動作を開始する $(T3_{SE_1}(a))$.

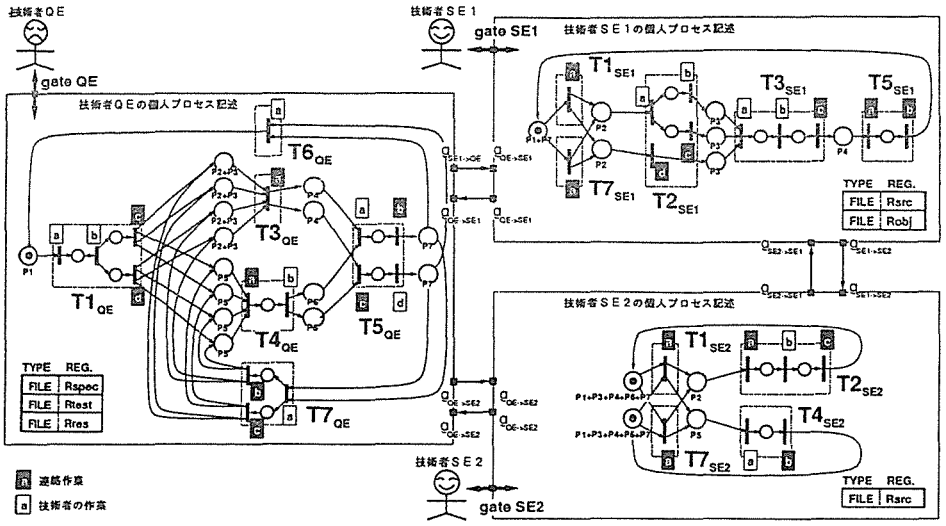
以上のような方針でサブネットを組み立て, 図 2 の $T2_{SE_1}$, $T2_{SE_2}$ を得る. なお, 技術者 QE は $T2$ の模倣に全く関係しない. この場合, 対応するサブネット $T2_{QE}$ は空としておく.

このように, 全体記述の各トランジションごとに責任者を定め, 責任者から始まるメッセージ系列により技術者間で同期をとり, そのトランジションの作業と同じ作業を協調して行う.

[Step2] 各技術者は全体記述と同形のネットを持つものとする. 各技術者はそのネットの各トランジションを対応するサブネットで置き換えることにより, その個人プロセス記述を得る. ここで, 各サブネットは一般に外部のプレースと接続するトランジションを複数個持つ (例えばサブネット $T2_{SE_1}$ はプレース $P3$ と接続するトランジションを3つ持つ) ため, 適当数のプレースを複製して, それらと接続する. また, サブネットが空である場合, これを除去する. これらのアルゴリズムの詳細についてはいずれも文献 [14] 参照.

3 プロセスの実行支援

前章では, 我々が提案した導出アルゴリズムの概略を述べた. 我々は, その手法にもとづき, 個人プロセス記述を自動導出するための自動生成システムを作成した.



技術者	トランジション		ラベル		
	サブネット	ID	ガード	入力／出力動作	レジスタ値更新式
QE	$T1_{QE}$	a	TRUE	$QE?x_1[x_1 = \mathrm{edit}(R_{spec})]$	
		b	TRUE	ι	$R_{spec} \leftarrow x_1$
		c	TRUE	$G_{qe \to sc_1}!M_1\{R_{spec}\}$	
		d	TRUE	$G_{qe \to sc_2}!M_2\{R_{spec}\}$	
	$T3_{QE}$	a	$w_1 == M_3\{R_{obj}\}$	$G_{sc_1 \to qe}?w_1$	
	$T4_{QE}$	a	$w_2 == M_4\{x_4\}$	$G_{sc_2 \to qe}?w_2$	
		b	TRUE	ι	$R_{test} \leftarrow x_4$
	$T5_{QE}$	a	TRUE	$SE_2?x_4[x_4 = \mathrm{edit}(R_{test})]$	
		b	TRUE	$G_{qe \to sc_1}!M_5\{\}$	
		c	$w_3 == M_6\{R_{obj}\}$	$G_{sc_1 \to qe}?w_3$	
		d	TRUE	ι	$R_{res} \leftarrow \mathrm{exec}(R_{obj}, R_{test})$
	$T6_{QE}$	a	$x_6 ==$"accept"	$QE!\mathrm{view}(R_{res})?x_6$	
	$T7_{QE}$	a	$x_7 ==$"feedback"	$QE!\mathrm{view}(R_{res})?x_7$	
		b	TRUE	$G_{qe \to sc_1}!M_7\{\}$	
		c	TRUE	$G_{qe \to sc_2}!M_8\{\}$	
SE_1	$T1_{SE1}$	a	$w_1 == M_1\{R_{spec}\}$	$G_{qe \to sc_1}?w_1$	
	$T2_{SE1}$	a	TRUE	$SE_1!\mathrm{view}(R_{spec})?x_2[x_2 = \mathrm{edit}(R_{src})]$	
		b	TRUE	ι	$R_{src} \leftarrow x_2$
		c	TRUE	$G_{sc_1 \to sc_2}!M_9\{x_2\}$	
		d	$w_2 == M_{10}\{\}$	$G_{sc_2 \to sc_1}?w_2$	
	$T3_{SE1}$	a	TRUE	$SE_1!\mathrm{view}(R_{src})$	
		b	TRUE	ι	$R_{obj} \leftarrow \mathrm{compile}(R_{src})$
		c	TRUE	$G_{sc_1 \to qe}!M_3\{R_{obj}\}$	
	$T5_{SE1}$	a	$w_3 == M_5\{\}$	$G_{qe \to sc_1}?w_3$	
		b	TRUE	$G_{sc_1 \to qe}!M_6\{R_{obj}\}$	
	$T7_{SE1}$	a	$w_4 == M_7\{\}$	$G_{qe \to sc_1}?w_4$	
SE_2	$T1_{SE2}$	a	$w_1 == M_2\{R_{spec}\}$	$G_{qe \to sc_2}?w_1$	
	$T2_{SE2}$	a	$w_2 == M_9\{x_2\}$	$G_{sc_1 \to sc_2}?w_2$	
		b	TRUE	ι	$R_{src} \leftarrow x_2$
		c	TRUE	$G_{sc_2 \to sc_1}!M_{10}\{\}$	
	$T4_{SE2}$	a	TRUE	$SE_2?x_4[x_4 = \mathrm{edit}(R_{test})]$	
		b	TRUE	$G_{sc_2 \to qe}!M_4\{x_4\}$	
	$T7_{SE2}$	a	$w_3 == M_8\{\}$	$G_{qe \to sc_2}?w_3$	

$M_n\{R\}$ は，メッセージ M_n がレジスタ R の値を含むことを意味する

図 2: 導出した各技術者 QE, SE_1, SE_2 の個人プロセス記述の例

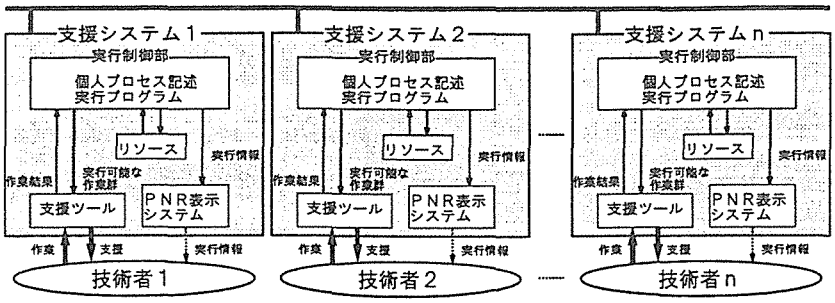


図 3: 実行支援システムの全体設計の概略図

本章では, 作成した自動生成システムと, 生成された個人プロセス記述を分散実行して技術者の作業を支援する支援システムの設計の概略を述べる.

3.1 各技術者のプロセス記述の自動生成システム

作成したシステムは, PNR モデルによるプロセスの全体記述と, レジスタとゲートの配置指定を入力とし, 同モデルで記述された各技術者の個人プロセス記述を出力する.

本システムでは, リソースの配置指定ファイルだけを別のものに変更することにより, その配置指定に対する各技術者の個人プロセス記述を得ることができる. このことから, ソフトウェアプロセスにおける技術者の増減や, リソースの配置状況の変更にも容易に対応できる.

3.2 プロセスの実行支援システムの設計

3.2.1 システムの全体設計

図 3 に実行支援システムの全体設計の概略図を示す. プロセスの実行支援は, 導出した個人プロセス記述をその技術者が使用する計算機上の実行プログラムに変換し, それを実行することで実現する. その実行プログラムは, 技術者が作業に必要な支援ツールの起動, 計算機内部でのリソース (レジスタ) の更新, 技術者間の連絡作業 (通信) , そしてそれらの実行順序制御を行う. また, 後述するプロセス表示システム (PNR 表示システム) に, プログラムの実行情報を与える.

実行プログラムは図 4で示す手順により, 個人プロセス記述と, どのツールを起動するかといった情報を作業者が定義するツールテーブル (後述) から生成する. したがって, それらから C 言語によるプロセスの記述を得るための PNR コンパイラ, プロセス情報を提供するためのプロセス表示システムが必要である. 以下ではこれらの設計について述べる.

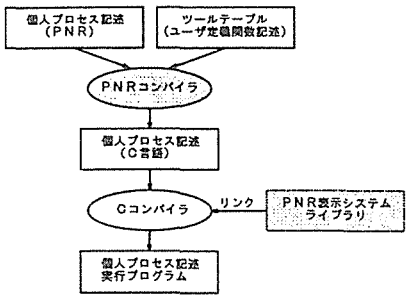


図 4: 実行制御部の生成

[PNR コンパイラ] 与えられた PNR による個人プロセス記述を C 言語による記述に変換する. C 言語での実現方法として, 基本的には実行可能なトランジションをすべて実行可能行列につなぎ, 行列の先頭から順次実行していく方法を用いるが, 一般にソフトウェア開発では, 例えば仕様書の編集やソースファイルの作成など, 一つの作業に時間がかかることが多い. そのため, プログラムの実行効率を考慮し, 例えばエディタによる編集作業を実行するトランジションは, 実際の編集作業中は終了待ち状態行列につないでおき, その間は実行可能行列の次のトランジションを実行し, 終了を知らせる入力があった時点で割り込みをかける一種のイベント駆動型のプログラムとして実現することにする. このようにすることにより, 並列に実行可能な作業において一方に時間のかかる作業がある場合, その終了を待つことなく, 他の作業の実行に移ることができる.

[プロセスの表示システム (PNR 表示システム)] 各技術者に, 個人プロセスの情報を与えるためのシステムである. 実行時の情報は, 実行プログラムから与えるため, したがって, プログラムが表示システムに実行時の情報 (現在のマーキングとレジスタ値) を渡せるように, 専用のライブラリを用意してコンパイル時にリンクする.

3.2.2 支援機能

上記の全体設計をもとに支援システムを作成するが，ここではそのシステムの支援機能についてそれぞれ詳しく述べる．

主な支援機能は以下の4つである．

1. プロセス情報の取得のためのプロセスの可視化機能
2. プロセスの実行制御および技術者間の連絡作業の自動実行機能
3. 作業の誘導およびそれに必要なツールの自動起動機能
4. プロセスの実行中断 (状態退避) および回復機能によるプロセスの動的修正機能

[1. プロセスの可視化] 各技術者は上述の PNR 表示システムにより，個人プロセスの情報を得ることができる．これにより，技術者は，何を並列に行えるのかなどといったプロセスの静的な情報だけでなく，今自分がどの作業をしているのか，また次に何をすべきかといったプロセスの動的な情報をも容易に把握することができる．さらに，どの連絡作業で遅延が生じているかといったことも管理者を通すことなくある程度把握でき，技術者間の非公式なコミュニケーション (電話，電子メールなど) により遅延を少なくすることもできる．

[2. プロセスの実行制御／連絡作業の自動実行] 各技術者が使用する計算機上で，個人プロセス記述から生成した実行プログラムを実行することにより，各技術者の作業の実行順序制御及び，計算機の通信システムを用いた連絡作業の自動実行を行う．これらにより，技術者は連絡作業の誤りを防ぐことができ，かつ本来の作業のみに集中できるため，個人あるいは全体の作業効率をあげることができる．

[3. 作業の誘導／ツールの自動起動] 現在のマーキングで発火可能なトランジションを発見し，そのようなトランジションから実行可能な作業を抽出し，それらを表示，選択できる機能を与える．さらに，選択した作業で必要となるツールを自動的に起動する機能を与える．この際，作業に必要なファイル等はツールの引数としてシステムが与えるため，作業者はツールを起動する手間が省けるとともに，作業の誤り (ファイルの読み込み誤りなど) を防ぐことができる．

また，ユーザの環境，開発するソフトウェアなどへの対応に柔軟性を持たせるため，edit() などにより実際に起動されるツールは以下に述べるツールテーブルに記述することにより自由に設定できるようにする．例えば，$QE\,?\,x_1[x_1 = \mathbf{edit}(R_{spec})]$ という作業に対し，ツールテーブルに $\mathbf{edit}(R_1)$: vi R_1 と書いておけば実行時にエディタ vi がファイル R_1 (内容は R_{spec}) を引数として起動されるが，これを $\mathbf{edit}(R_1)$: emacs R_1 とすることでエディタ emacs が起動できるようになる．さらに現実の作業内容を考慮し，全体記述における1つのツールの指定により実行時に複数個のツールを起動できるようにする．例えば $QE\,?\,x_1[x_1 = \mathbf{maketex}(R_{spec})]$ という作業に対し，maketex を LaTeX の文書作成のための一連の作業 (エディタと jlatex，プレビューア，プリンタ出力コマンドなどの繰り返し) として用いたい場合がある．このとき，

$\mathbf{maketex}(R_1)$:

R_{tmp1} = emacs R_1 ;
R_{tmp2} = jlatex R_{tmp1} ;
xdvi R_{tmp2} ;
R_{tmp3} = jdvi2kps R_{tmp2} ;
lpr R_{tmp3}

といったように，必要なコマンドとそれらの依存関係をテンポラリレジスタを用いてすべて記述する．そして，実行時にこれらのなかから必要なコマンドをメニュー形式で繰り返し選択でき，作業者がメニューの終了ボタンを押したときの結果が x_1 として入力できるようにする．このように，作業者の好みや，作業の内容に柔軟に対応できる機能を与える．

[4. プロセスの動的修正] 大規模なソフトウェアの開発などでは，スケジュールの変更など開発時の工程の変更が頻繁に起こり得る．これはソフトウェアプロセスの実行時におけるプロセス自身の変更が頻繁に起こり得るを意味し，これに対応できる機能を備えることが望ましい．

設計するシステムでは，変更が必要となった時点での各計算機でのレジスタ値を退避させる．そしてある技術者 (例えばスケジュール管理者など) がプロセスの変更に対応した新しい全体記述を作成し，その記述から自動生成システムを用いて各技術者の新しい個人プロセス記述を導出し，各技術者に転送する．各技術者は新たな制御部を生成して実行を再開する，といった方法で，これに対応する．

そのためには，修正が必要となった時点でのレジスタ値を退避し，実行を中断する必要があるが，サブネットの実行途中で停止させた場合，技術者間でのレジスタ値の整合性がとれていない場合がある (例えば，$T2$ において R_{src} の値を SE_1 は更新済み，SE_2 は未更新である場合)．そこで，全体仕様のあるトランジション T に対し，個人記述レベルでは T の責任者以外の技術者は T の責任者からのメッセージがなければ動作を開始できないことを利用し，ある T をその責任者が実行可能となった時点でシステムの実行を停止し，他の技術者にそのことをメッセージで伝えればよい．また，同時に停止時のマーキング情報をプロセス修正者に伝える．例えば，$T2$ に対応するサブネットを実行中であれば，その次のトランジション $T3$ の責任者 SE_1 がすべてのプレース $P3$ にトークンが与えられた時点で制御部の実行の停止を他の技術者 QE，SE_2 に伝える．また，プロセス修正者が QE であったとすれば，その時のマーキング情報を同時に QE に伝える．

そしてプロセス修正者に対しあらかじめ指定した編集

ツールなどを起動してプロセスの全体記述の修正 (新しい全体記述の作成) を促す. この修正者は, 停止時のマーキング情報から, 新しいマーキング (どこから実行を再開するかの情報) を定める. 修正後は自動生成システムを起動し, 新しい各技術者の個人プロセス記述を導出する.

最後に導出した個人プロセス記述を各技術者に転送し, 各技術者はその記述から新しい制御部を生成し, レジスタ値を復元して新しいマーキングから実行を再開する.

このように, 動的修正の必要が生じた場合, 技術者間のレジスタ値の整合性を失うことなく制御部の実行を中断し, 自動生成システムと同調して新しいプロセス記述を得た後, それを実行再開する機能を付加する.

4 おわりに

本稿では, 文献[14] で提案した, レジスタ付ペトリネットモデルで記述されたソフトウェアプロセスの全体記述と, 各技術者へのレジスタやゲートの配置指定から, 各技術者の個人プロセス記述を導出する手法をもとに, それらの個人プロセス記述を自動導出するシステムと, 導出した個人プロセス記述を分散実行する支援システムの設計の概略について述べた. 実行支援系を用いることにより, 各技術者は他の技術者との連絡作業に煩わされることがなくなる. また実行可能な作業の誘導や作業状況の把握により, 作業を効率よく進めることができる. さらに, 作業環境やスケジュール, 作業内容の変更など, ソフトウェア開発過程で起こり得る様々な問題にも十分対処できると考える.

今後の課題として, システムの完成を目指すと共に, 支援システムにリアルタイムスケジューリング機構を持たせたい. 我々は, 文献 [14] のアルゴリズムにおけるペトリネットモデルのクラスを, 全体記述に時間制約を与えることができるように拡張することを検討中である. この手法を考案, 適用することにより, より現実的な支援システムの実現を目指す.

参考文献

[1] Curtis, B., Kellner, M. and Over, J. : "Process Modeling," Commun. ACM, Vol. 35, No. 9, pp. 75-90 (1992).

[2] Osterweil, L. J. : "Software processes are software too," Proc. 9th Int. Conf. on Software Engineering (ICSE-9), pp. 2-13 (1987).

[3] Iida, H., Mimura, K., Inoue, K. and Torii, K. : "Hakoniwa: Monitor and Navigation System for Cooperative Development Based on Activity Sequence Model," Proc. 2nd Int. Conf. on Software Process (ICSP-2), pp. 64-74 (1993).

[4] Inoue, K., Ogihara, T., Kikuno, T. and Torii, K. : "A Formal Adaptation Method for Process Descriptions," Proc. 11th Int. Conf. on Software Engineering (ICSE-11), pp. 145-153 (1989).

[5] Barghouti, N. S. : "Supporting Cooperation in the MARVEL Process-Centered SDE," ACM SIGSOFT, Vol. 17, No. 5, pp. 21-31 (1992).

[6] Deiters, W. and Gruhn, V. : "Managing Software Processes in the Environment MELMAC," ACM SIGSOFT, Vol. 15, No. 6 (1990).

[7] Bandinelli, S., Fuggetta, A. and Grigolli, S. : "Process Modeling in-the-large with SLANG," Proc. 2nd Int. Conf. on Software Process (ICSP-2), pp. 75-83 (1993).

[8] Yasumoto, K., Higashino, T. and Taniguchi, K. : "Software Process Description using LOTOS and Its Enaction," Proc. 16th Int. Conf. on Software Engineering (ICSE-16), pp. 169-179 (1994).

[9] Murata, T. : "Petri Nets: Properties, Analysis and Applications," Proc. IEEE, Vol. 77, No. 4, pp. 541-580 (1989).

[10] Kellner, M. et al. : "ISPW-6 Software Process Example, " Proc. 1st Int. Conf. on the Software Process (ICSP-1), pp. 176-186 (1991).

[11] 松浦佐江子, 本位田真一 : "ソフトウェアプロセスにおける協調とその抽象化について," コンピュータソフトウェア, Vol. 10, No. 2, pp. 48-64 (1993).

[12] 安本慶一, 東野輝夫, 谷口健一 : "LOTOS によるソフトウェアプロセスの記述とその実行," コンピュータソフトウェア, Vol. 12, No. 1, pp. 16-30 (1995)

[13] Yamaguchi, H., Okano, K., Higashino, T. and Taniguchi, K. : "Software Process Description in a Petri Net Model and It's Distributed Execution," 信学技報, Vol. 94, No. 334 (SS94-38), pp. 25-32 (1994).

[14] Yamaguchi, H., Okano, K., Higashino, T. and Taniguchi, K. : "Synthesis of Protocol Entities' Specifications from Service Specifications in a Petri Net Model with Registers," Proc. 15th Int. Conf. on Distributed Computing Systems (ICDCS-15), pp. 510-517 (1995).