ソフトウェア工学 122−17
（1999. 3. 19）

# カラーペトリネットに基づくソフトウェアアーキテクチャ記述

Wu Wenxin　　佐伯 元司

東京工業大学 情報理工学研究科 計算工学専攻

〒152-8552 東京都目黒区大岡山 2-12-1
E-mail : wenxin@cs.titech.ac.jp

あらまし　アーキテクチャ設計は要求と実装とのギャップを埋めるのに重要な役割を果たしている．また，システムのパフォーマンス，信頼性など非機能的な要求の実現とも密接に関連している．従って，設計したアーキテクチャを形式的に記述し，その記述に対して解析や評価を行うことにより，ソフトウェア開発のより早い段階で非機能的な要求を実現できるかどうかを調べることができ，開発の生産性を向上させることができる．本稿では，システムの構成と挙動を評価するため，カラーペトリネットを意味的基礎とする記述を提案し，アーキテクチャ記述言語の開発を行った．また，記述の再利用についても考察を行った．

キーワード　ソフトウェアアーキテクチャ，カラーペトリネット

# A Technique to Specifying Software Architectures Based on Coloured Petri Nets

Wenxin Wu　　Motoshi Saeki

Department of Computer Science,
Graduate School of Information Science & Engineering,
Tokyo Institute of Technology

2-12-1 Oookayama, Meguro-ku, Tokyo 152-8552, Japan
E-mail : wenxin@cs.titech.ac.jp

Abstract　The quality of an architectural design has a great influence on achieving non-functional requirements to the system, so formal evaluation and validation techniques to designed architectures are necessary in the early phase of development processes. In this paper, we present a new description language based on Coloured Petri Nets (CPN) and a technique for reusing architectural constituents. We use CPN as the underlying semantic theory of our description language, so that evaluation and analysis on the architectural descriptions can be made to validate the acceptability of designed architectures.

Key words　Software architecture, Coloured Petri Nets

# 1 はじめに

ソフトウェア開発のライフサイクルにおいて，アーキテクチャ設計は要求と実装とのギャップを埋めるのに重要な役割を果たしている [1]．ソフトウェアアーキテクチャはシステムの構造のみならず，システムのパフォーマンス，信頼性など非機能的な要求の実現とも密接に関連している．従って，設計したアーキテクチャを形式的に記述し，その記述に対して解析や評価を行うことにより，ソフトウェア開発のより早い段階で非機能的な要求を実現できるかどうかを調べることができ，開発の生産性を向上させることができる．

現在，形式的な記述法として，Z(集合論 + 述語論理) [2], CCS, CSP [3] [4], $\pi$-計算 [5], Abstract Chemical Machine [6], 半順序イベント集合を意味的基礎とするアーキテクチャ記述法がすでに提案されている．しかし，1) アーキテクチャを形式的に記述する方法論が確立されておらず，形式的な記述に慣れていない実務者にとって実際に実用規模のシステムのアーキテクチャを記述するのは難しい，2) 記述の実行やシミュレーションなど解析支援系の能力が不十分であるため，アーキテクチャ記述を解析するのに適切ではない，という欠点がある．

そこで，本稿ではカラーペトリネット (CPN) [7] に基づいた記述法を提案する．この方法では，まず CPN を記述の意味的基礎とする．その理由として，CPN はシステム挙動に関する解析や評価が行え，CPN に関するシミュレーションや解析の手法と強力なツールがすでに開発されているからである．しかし，CPN はほかの形式的言語と同じく，慣れていない人にとっては，複雑な機能が多くあるため，記述が難しいという問題がある．そこで，本研究は簡単に記述できるアーキテクチャ記述言語 (ADL) を開発し，ADL による記述を CPN へ自動変換するツールを開発した．さらに，再利用技術に基づいた記述法を提案した．再利用の目標は，あらかじめ定義された部品の組合せのみでアーキテクチャを記述できることである．

以下の節では，次のことを述べる．まず，2 節でアーキテクチャ記述を説明し，提案する手法の概要を述べる．3 節で CPN について概説し，CPN によるアーキテクチャの記述法を述べる．4 節でアーキテクチャスタイルに基づいた再利用部品の抽出について述べる．5 節で開発した ADL についての説明を行う．6 節で手法を黒板システムに適用し，手法の有効性について考察する．

# 2 ソフトウェアアーキテクチャ記述

## 2.1 ソフトウェアアーキテクチャとアーキテクチャ記述言語

ソフトウェアアーキテクチャはシステムの構成要素と構成要素間の関係を定義したものである．アーキテクチャモデルは主に以下の構成要素から成っているとされる [1].

- コンポーネント：計算またはデータ貯蓄のユニット
- コネクタ：コンポーネント間の相互作用をモデル化したもの
- インターフェース：コンポーネントとコネクタ，コンポーネントと外部との相互作用点の集合

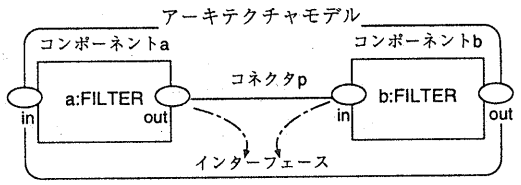



図 1: パイプ／フィルタ型アーキテクチャモデル

図 1 に 2 つのコンポーネントと 1 つのコネクタから構成されるパイプ／フィルタ型アーキテクチャを示す．このアーキテクチャは 2 つのコンポーネント (コンポーネント “a” とコンポーネント “b”) とそれらを連結するコネクタ “p” から構成される．また，図にある小さい楕円に表わされるインターフェースはコンポーネントとコネクタの連結点を表わしている．コンポーネント “a” は，インターフェース “in” より外部からのデータを受け取り，その後インターフェース “out” に処理したデータを出力する，“out” に連結されたコネクタがそこからデータをコンポーネント “b” のインターフェース “in” に渡す．“b”はまた処理結果をこのアーキテクチャの出力インターフェースともなるインターフェース “out” に渡す．

ソフトウェアアーキテクチャを形式的に記述し，記述に対するシミュレーションを行うことによって，設計の誤りを検出することが考えられる．現在いくつかのアーキテクチャ記述言語がすでに開発されている．そのうち，システムアーキテクチャのプロトタイピングのために設計され，並行動作イベントを定義し，シミュレーション実行を目的とした Rapide，CSP を意味的基礎とした Wright，$\pi$ 計算に基づくプロセス代数 FSP

を意味的基礎としたDarwinなどが挙げられる[8]．しかし，それらのいずれもイベントの生起確率を記述できず，パフォーマンス評価など非機能的特性をチェックする能力は不十分である．

### 2.2 アーキテクチャ記述言語への要求

シミュレーションによりパフォーマンス評価を行うため，アーキテクチャ記述言語に対して次に示す要求を満たさなければならない．

- **形式的意味論と実行可能**
  記述されたアーキテクチャに対する解析を計算機上で行うためには，記述が形式的な意味を持つべきである．また，本手法ではシミュレーションを用いて記述のチェックを行うので，この記述言語の基礎となる形式意味論は操作的に与えられる，すなわち，実行可能でなければならない．
- **システム動作の記述**
  アーキテクチャ設計は大規模なシステムを対象としており，特に並列システムを対象することが多い．そのようなシステムのアーキテクチャ設計を評価するため，アーキテクチャ言語が並列動作と同期を記述できる必要がある．
- **シミュレーション用データの記述**
  シミュレーションを行うには，アーキテクチャ記述に実行時間，フォールト発生率などシミュレーションするためのパラメータを記述できることが要求される．
- **解析理論やツール**
  システムの解析や評価を行うための理論，効率的な解析や評価アルゴリズム，ユーザインターフェースの優れた支援ツールが不可欠である．

### 2.3 提案する記述手法

本手法は，上述の要求を満たすため，カラーペトリネットを意味的基礎とし，記述の再利用ができるような記述法を提案する．本手法に基づくアーキテクチャ記述は以下の手順に従って行われる．

1. 要求仕様に基づき，本研究で開発したアーキテクチャ記述言語 (ADL) でアーキテクチャを記述する．また，設計者が再利用データベースから部品や型を抽出して利用することができる．
2. アーキテクチャ記述をCPN変換器に入力し，変換器はそれをCPNへ変換する．
3. 変換したCPNを，CPN既存のツールであるDesign/CPN [9] に入力し，シミュレーションと解析を行う．

## 3 カラーペトリネットによるアーキテクチャ記述

### 3.1 カラーペトリネット

カラーペトリネット (CPN) は，より複雑なシステムを記述できるようにペトリネットを拡張した手法である．CPNにおいては，“プレース”，“トランジション”と“トークン”のほかに，“カラー”，“ガード”と“式”といった概念が導入されている．CPNにおけるトランジションの発火条件は以下になる．

- そのトランジションの入力プレースに少なくとも1つのトークンがある．
- 入力プレースにあるトークンはそのプレースからトランジションに至るアークに付加された式を満たす．
- 入力プレースにあるトークンはトランジションが持つガード条件を満たす．

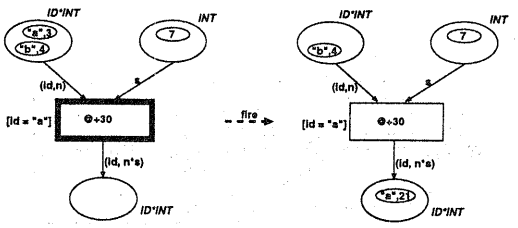



図 2: CPNの例

図2にCPNの1つの例を示す．楕円とボックスはそれぞれプレースとトランジションを表わす．プレースにある小さい楕円はトークンを表わす．トークンはカラーを持つ．このカラーというのは，トークンが持つ属性である．CPNにおけるプレースは，カラーの型 (カラーセット) を持っていなければならない．例えば，あるプレースはカラーセット“INT”を持つとすると，そのプレースに入り得るトークンのカラーは“INT”型でなければならない．図2では，式“(id, n)”と“s”がトランジションに至るアークに付加されている．“(id, n)”はカラー変数の組み合わせである．式“[id = “a”]”は

そのトランジションのガードである．トークン (“a”,3) の場合にこの式が成り立つので，トランジションが発火する．発火のあと，出力アークに付加されるカラー変数 “(id, n)” に値 “(“a”,21)” がバインドされる．式 “@+30” はそのトランジションの発火が 30 ユニット時間かかることを意味する．

### 3.2 CPN によるアーキテクチャ記述

本手法では，CPN を記述の意味的基礎としている．アーキテクチャモデルの各要素およびモデルに関する評価を行うための記述と CPN との対応を表 1 に示す．これらの対応づけに基づき，図 1 のフィルタ／パイプスタイルのアーキテクチャを，CPN へ変換した結果は図 3 になる．この CPN 記述は元のシステムと同じ挙動を持つことがわかる．

| アーキテクチャ概念 | CPN |
|---|---|
| コンポーネント | CPN |
| コネクタ | CPN |
| インターフェース | 型付きプレース |
| コンポーネントとコネクタの接続 | インターフェースとなるプレースが同じプレースであると見なす |
| コンポーネントの実行時間 | トランジションかアークに付化される発火に要する時間 |
| フォールト発生率 | トランジションの発火確率 |

表 1: アーキテクチャ概念と CPN との対応づけ

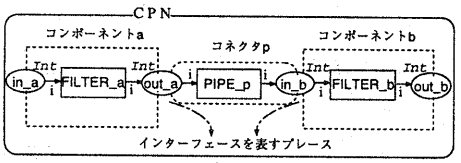



図 3: パイプ／フィルタ型アーキテクチャの CPN による記述

以上の対応により，カラーペトリネットによるアーキテクチャ記述に対して，以下のシミュレーションと解析を行うことができる．

- アーキテクチャ要素間の接続の役割を担うインターフェースは型付きプレースで表すので，接続相手を制限でき，構文的なミスマッチを防ぐ．
- コンポーネントやコネクタに処理時間を設定することにより，システムのボトルネックとなっている箇所を検出できる．
- フォールト発生確率を与えることにより，どこで生じるフォールトが全体に一番大きな影響を与えているかに関する解析ができる．
- カラーペトリネットにおける活性という性質を判定するアルゴリズムによって，あるコンポーネントの入力インタフェースにデータか制御が置かれると，そのコンポーネントの少なくとも 1 つの出力インタフェースに必ずデータが出力されることを保証できるかどうかをチェックすることができる．

## 4 アーキテクチャスタイルと再利用

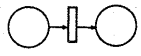
アーキテクチャ記述の意味的基礎として，CPN を利用することが有効であるが，CPN はほかの形式言語と同様，未定義部分がないように詳細レベルまで正確に記述する必要がある．また，本手法の目的であるシステムのパフォーマンス評価に関係しない機能も多く含んでおり，設計者がそれらをすべて習得するのはコストの浪費ともなり，形式手法を有効に利用することができない．そのため，本研究では次節で説明するアーキテクチャ記述言語 (ADL) を開発し，また，記述に再利用技術を導入する．

再利用技術の導入により，設計者が既存の記述部品の組み合わせを記述するだけで，アーキテクチャ記述ができることが望ましい．そのため，アーキテクチャを構成する基本的な要素を再利用可能な部品として用意し，それら部品間の組み合わせやカスタマイズ方法をルール化することが考えられる．

しかし，どんな部品がアーキテクチャを構成する基本的な部品であるかが問題になってくる．そこで，本研究はまずシステム設計によく現われるパターン，アーキテクチャスタイル [1] に注目した．

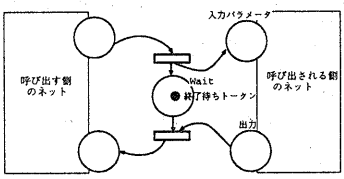
CPN によるアーキテクチャスタイルの記述とその考察から，アーキテクチャスタイルはさらに基本的な部品から構成される場合が多いことが判明した．アーキテクチャスタイルの中で共通に出現しているいくつかの部品の概要および対応する CPN による記述を以下に示す．

- **データフロー:**
  パイプ／フィルタスタイルなどに現れるような，データの流れによってコンポーネント間を連結するコネクタである．

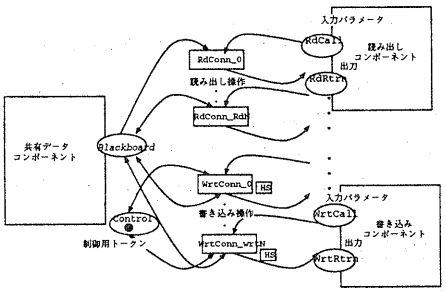


- **手続き呼び出し:**
  このコネクタにおいては，呼び出し側は手続きの

実行が終了し，実行結果が戻されるまで，ほかのすべての動作を中止することが要求される．また，1つのコンポーネントの手続きが複数のコンポーネントに呼び出される可能性があるので，結果を正しく呼び出す側に戻す必要がある．本研究では，下図のように，プレース “Wait” を設け，手続き呼び出しの要求がくると，そのプレースにトークンを置くことによって，上記の要求が満たされることを保証する．




- **排他制御:**
  共有データへの同時書き込みを禁止するコネクタである．下図のように，書き込み操作を表わすトランジションに連結するプレース “Control” を設ける．そのプレースはただ1つのトークンを持っている．それによって，共有データへの書き込みはどちらかのコンポーネント1つしか行うことができないことを保証する．




そのほかに，以下のものも基本型部品となる．

- イベント駆動システムに現れる手続きを暗黙に呼び出す暗黙呼び出しコネクタ
- 同期通信
- 非同期通信

以上述べた基本型部品を本研究で用意した**再利用部品データベース**に格納する．それらの基本型部品を用いて，以下列挙した Mary Shaw ら [1] が定義したアーキテクチャスタイルを記述できることを確認した．

- パイプ／フィルタスタイル
- レイヤシステム
- 抽象データ型とオブジェクト指向構成
- イベント駆動，暗黙起動スタイル
- 黒板スタイル
- クライアント／サーバシステム
- インタープリタスタイル

部品間の結合を再利用できるようにするために，基本型部品から構成されたアーキテクチャスタイルを複合部品として用意する．設計者は再利用部品データベースから部品を抽出し，部品をカスタマイズしたり組み合わせたりすることによって，アーキテクチャ記述を構築できる．もしそのアーキテクチャが汎用性があるものであれば，その記述をデータベースに追加することができる．

# 5 CPN に基づく記述言語

前節で述べた部品および部品間の関係を記述するには，CPN をそのまま用いるのは実務者にとっては難しい．本研究では，CPN に基づいたアーキテクチャ記述言語 (ADL) を開発した．その ADL においては，図による表記法 (Graphical Notation) とテキストによる記述 (Textual Notation) の両方を用意した．ここでは，パイプ／フィルタ型のアーキテクチャを例として用い，それらの概略を述べる．

## 5.1 Graphical Notation

Graphical Notation は設計を可視化し，設計者がより容易にシステムアーキテクチャを理解できる利点がある．本研究で開発した Graphical Notation では，コンポーネント，コネクタとインターフェースをそれぞれボックス，楕円ボックスと楕円で表す．パイプ／フィルタ型アーキテクチャ(図 1) を Graphical Notation で記述したものを図 4 に示す．この図では，コンポーネント “a” に書かれている “a:FILTER” はコンポーネントの名前が “a” で，型が “FILTER” であることを表している．また，インターフェースに書かれている “O:i2:Int” はこのインターフェースが出力インターフェースであり，名前と型はそれぞれ “i2” と “Int” であることを表している．インターフェースの型はそのインターフェースに入れるデータの型を意味する．さらに，“@+30” はコンポーネント a の実行に 30 ユニット時間がかかるということを意味し，“#95%” はコンポーネント b における正常に稼働する確率は 95%，すなわち，フォールト発生確率は 5% であることを意味する．

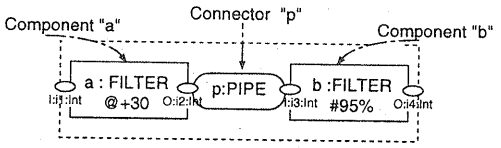



図 4: パイプ／フィルタ型アーキテクチャの図による記述

### 5.2 Textual Notation

Textual Notation を用いて，より抽象的なレベルでシステムを記述できる．また，部品の内部の記述，つまりどのように動作するかの記述は，本研究で定義したカラーペトリネットのテキスト表現 (CPN-like 言語) によって行われる．本研究でそれを採用する目的は，テキストによる ADL にこの記述を埋め込むことにある．

Textual Notation では “interface”, “component_ins”, “connector_ins”, “binding” などの識別子をキーワードとして用意し，任意個のフィルタから構成されるパイプ／フィルタスタイルを次のように記述することができる．この記述にあるパラメータはそれぞれ，コンポーネント (フィルタ) の数 (n)，フィルタの型 (FILTER_TYPE) およびパイプの型 (PIPE_TYPE) を意味する．

```
component Pipe_Filter_Style (
       int n, <FILTER_TYPE>, <PIPE_TYPE>){
  interface
    I : in : INPUT;
    O : out: OUTPUT;
  component_ins
    for i: 0..(n-1) {
      f[i] : FILTER_TYPE;
    }
  if (n >= 2){
    connector_ins
      for i: 0..(n-2) {
        p[i] :PIPE_TYPE;
      }
    binding
      in -- f[0].in;
      for i: 0..(n-2) {
        f[i].out -- p[i].in;
        p[i].out -- f[i+1].in;
      }
      f[n-1].out -- out;
  }
}
```

このスタイルは “FILTER_TYPE” 型のコンポーネントインスタンスの配列 ([]) と “PIPE_TYPE” 型のコネクタ配列から実装されている．複製構成子 (replicator construct) **for** を用いて，実際のインスタンス数とそれらがどのように結合されているかの宣言を行う．構成子 **if** の記述で，条件に応じて，コンポーネント／コネクタのインスタンスを選択的に結合することができる．さらに，コンポーネントとコネクタとの接続は，それらのどのインターフェース同士を連結するかを **binding** セクションに記述することによって定義する．

ここで，コンポーネントに関してのテキストによる表記法をつぎのようにまとめる．

```
component ComponentTypeName([Parameters]) {
 declaration
  /* カラーセットか型の宣言                    */
  interface
   Direction(IN | OUT | IO):
              Interface Name : Interface Type;
  component_ins
    Component instance name : Type;
   |Component instance name = Type([Parameters]);
  connector_ins
    Connector instance name : Type;
   |Connector instance name = Type([Parameters]);
  binding
   [Instance Name.]InterfaceName
                -- [InstanceName.]InterfaceName;
  cpn_spec
  /* 基本型の CPN-like 言語による記述          */
  exec_attribute
    InstanceName.execution_time = 実行時間;
    InstanceName.execution_rate = 正常稼働確率;
}
component ComponentInstance
          = ComponentTypeName([Parameters]);
```

一方，コネクタはコネクタのみからしか構成されないため，コネクタの表記法はこのコンポーネントの表記法に比べると，コンポーネントインスタンスを宣言するセクションを含まない点を除いて同じである．

## 6 適用例

### 6.1 黒板システム

本論文では，黒板システムを提案した手法への適用例として取り上げる．黒板スタイルは，黒板コンポーネントと Knowledge Source コンポーネントという二種類のコンポーネントと，制御コネクタから構成される．その基本的な特徴は以下になる．

**黒板コンポーネント** は共有データを表す．

**Knowledge Source コンポーネント** は共有データにアクセスし，それぞれ独立して並列に動いているプロセスを表す．共有データの読み出し，書き込みという二種類のコンポーネントがあると考えられる．

制御コネクタ は黒板コンポーネントと Knowledge Source コンポーネントを接続するものであり，要求するデータが正しく要求した側に戻されるのを保証し，また共有データへの同時書き込みを禁止する役割を果たしている．

ここで，対象とする具体的なステムは，3 つの Knowledge Source コンポーネント (“rd”, “wrt0” と “wrt1”) を含むシステムである．そのうち，“rd” は共有データを読み出し操作を行うが，“wrt0” と “wrt1” は共有データへ書き込み操作を行うとする．

この黒板システムを 5 節で提案した ADL の Graphical Notation に従った記述を図 5 に示す．ここで，制御コネクタ (“Blackboard_Connector_Ins” 型) は再利用部品データベースに格納されている既存の基本部品であるとする．

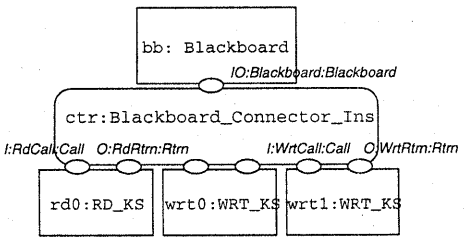



図 5: Graphical Notation による黒板インスタンスの構築

## 6.2 再利用に基づくスタイルの記述

黒板スタイルの CPN による記述の考察から，制御コネクタを，その機能により，手続き呼び出しコネクタ，排他制御コネクタから構成することを考える．まず，排他制御コネクタを次のようにカスタマイズする．ここで，“EXCLCTRL” は排他制御コネクタの型名であり，パラメータとなる “1” と “2” は読み出しコネクタと書き込みコネクタがそれぞれ 1 個と 2 個であることを表わす．

$connector\ excl = EXCLCTRL(1,2)$

そして，図 6 のように，手続き呼び出しコネクタと連結する．

## 6.3 シミュレーションと解析

上記の黒板システムにおいて，Knowledge Source コンポーネントと排他制御コネクタを連結する 3 つの手続き呼び出しコネクタ ($Connector_{read}$, $Connector_{wrt1}$ と $Connector_{wrt2}$ ) が通信のためそれぞれ 15ms, 200ms, 10ms を費やすと仮定する．また，アクセス要求をそれぞれ 5 回，7 回，10 回ランダムに発生させる．以上のシミュレーション用データを ADL による

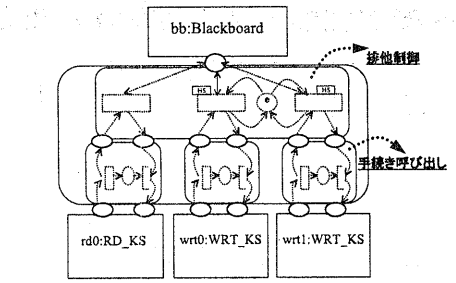



図 6: 制御コネクタの構成

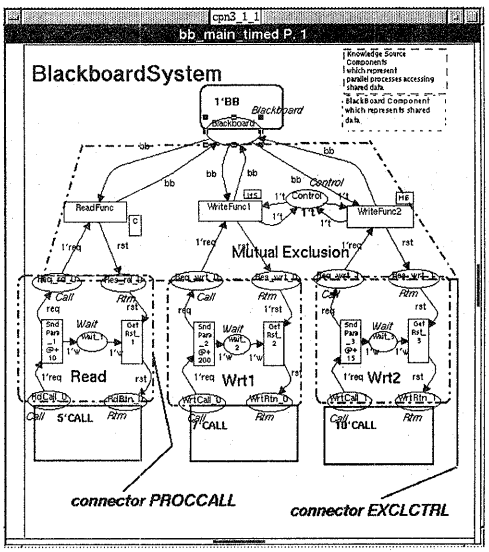



図 7: 黒板システムのシミュレーション例

記述に追加し，記述を CPN へ変換した結果を図 7 に示す．また，その条件で行ったシミュレーションの結果を図 8 に示す．ここでは，ある特定の時間で，どのコネクタが実行しているかを示している．横軸は時間で，縦軸はコネクタの名前である．(システム起動時刻は “0” であると仮定する)．図 8 により，コネクタ

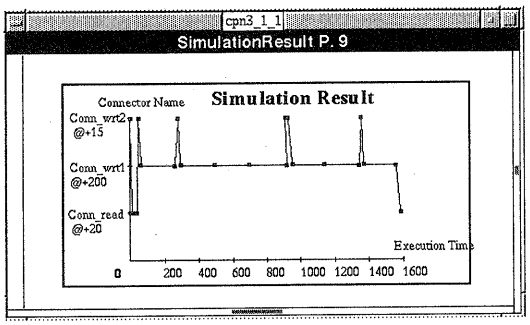



図 8: 黒板システムのシミュレーション結果

$Conn_{wrt1}$がもっとも多くの実行時間を費やしていることが明らかである．従って，パフォーマンスの視点から，そのプロセスがシステムのボトルネックとなる可能性が大きいことがわかった．これにより，ボトルネックの可能性が視覚的に得られたことで，本手法はアーキテクチャ設計がパフォーマンス評価など非機能的な要求を実現しているかどうかをチェックすることに有効であることを確かめた．

さらに，システムのコンポーネントかコネクタにフォールト発生確率を与えることによって，どの部分のフォールトがシステム全体のパフォーマンスにどのくらいの影響を与えるかを調べることができる．そのほか，CPN の解析ツールによって，Occurrence graph を算出し，Occurrence graph 上のデッドノードを検出し，そのデッドノードに至るノードをたどれば，どのようにしてシステムが止まったかを検出できることがすでに確認されている [7].

しかし，ここで与えたシミュレーションパラメータは仕様や経験から推測した実行時間であるので，より正確な予測を立てるために経験に基づいたシミュレーションモデルの蓄積が必要である．

## 7 おわりに

本研究では，カラーペトリネットを意味的基礎とする記述法を提案した．カラーペトリネットが持っているシミュレーション能力や解析能力を生かすことによって，ボトルネックの検出などのパフォーマンス評価ができる．また，実務者にとって形式的に記述するのが難しいという問題を解決するために，アーキテクチャスタイルに基づいた記述の再利用法を提案し，CPN の front-end とするアーキテクチャ記述言語とそれの変換ツールを開発した．

今後の課題としては，再利用できる部品の収集，動的変化の記述およびアーキテクチャ設計方法論の確立などが考えられる．

## 参考文献

[1] M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall, 1996.

[2] G. Abowd, R. Allen, and D. Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pp. 9–20, December 1993.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. In *ACM Transactions on Software Engineering and Methodology*, 1997.

[4] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Software Engineering*, Italy, May 1994.

[5] J. Magee, J. Kramer, and D. Giannakopoulou. Software architecture directed behaviour analysis. In *Proceedings of Ninth International Workshop on Software Specification and Design*, 1998.

[6] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Software Eng.*, Vol. 21, No. 4, April 1997.

[7] K. Jensen. *Coloured Petri Nets*, Vol. 1-2. Springer-Verlag, 1992.

[8] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description language. *Software Engineering Notes*, Vol. 22, No. 6, pp. 61–76, 1997.

[9] *Design/CPN Reference Manual for X-Windows Version 2.0.* Meta Software Corporation, 1993.

[10] Wu Wenxin. カラーペトリネットに基づくソフトウェアアーキテクチャ記述. ウィンターワークショップ・イン・高知 論文集, pp. 7–8. 情報処理学会, 1999. 情報処理学会シンポジウムシリーズ Vol.99, No.1.