ソフトウェア工学 120－16
（1998．7．24）

ソフトウェアアーキテクチャに基づく性能問題へのアプローチ

野田夏子，岸 知二

NECソフトウェアデザイン研究所

<概要>
ソフトウェア開発においては、一義的な機能の実現だけでなく、性能等の非機能的特性に関しても厳しい要求が課せられる。しかし、設計時において最終的なソフトウェアの性能を予測することは難しく、実際の開発では性能問題が多出している。我々はこうした性能問題を回避するために、部分的に実装・実測を行いその結果に基づいた全体の性能の見積もりを設計に利用する手法を検討している。本稿では、こうした実測による性能問題へのアプローチを機能させるために考慮すべき問題をソフトウェアアーキテクチャの観点から明らかにするとともに、ソフトウェアアーキテクチャを表現するグラフの導入と、このグラフを設計に活用する手法の提案を行う。

An Approach to Performance Issues based on Software Architecture

Natsuko Noda, Tomoji Kishi

Software Design Laboratories
NEC Corporation

<abstract>.
Non-functional properties such as performance or reliability are determined by many factors, and we can hardly understand the relationships between these factors and the properties. Instead of completely understand the nature of non-functional properties, we are examing the design method for finding out ``safer'' way to attain required non-functional properties, based on partial knowledge about the non-functional properties of components. For that purpose, we introduce, in this paper, collaboration/dependency graph, which represents the dependency between components, and explain how we find more suitable chunk of components to catch properties of whole software using this graph.

1 はじめに

ソフトウェア開発においては、一義的に要求される機能の実現が求められるとともに、性能や信頼性といった非機能的特性に対しても様々な要求が課せられ、その達成が求められる。

特に、性能に関しては、レスポンスタイムやスループットについて一定の値を保証することが求められるなど、厳しい要求が課せられる場合がある。しかし、このような性能に関して開発中に予測することは難しく、結局期待される性能が得られないなどの性能問題が、実際に多発している。

性能問題を回避するためには、通信時間など性能面に関りそうな部分だけを実装し実測して、それを基に全体の性能を見積るなどの方法が現実的によく行われている。しかし、こうした見積りが常にうまくいっているとは限らない。その理由として、実測対象とする部分の性能が最終的なシステムの性能とどのような関わりを持つかについての体系だった理解や検討がなされないまま、アドホックにこのような部分的な実装・実測が行われている等の理由が指摘できる。従って部分的な実測に基づいて全体の性能の見積もりを行う際には、部分的な実測の結果がソフトウェア全体の性能の捕捉に有効に活用できるように実測する部分を選定することが重要となる。

本稿では、要求される性能を達成するように設計を行う際に部分的な実測結果を効果的に活用することを目的として、性能面からのソフトウェアアーキテクチャを捉えるとともに、性能面のソフトウェアアーキテクチャを表現する手段として協調/依存グラフを導入し、このグラフを実測部分の判断に活用する手法を提案する。なお、以降、特に断らない限りアーキテクチャとはソフトウェアアーキテクチャを指すこととする。

2 性能問題と性能設計

ソフトウェアは、一義的に要求される機能を実現しなければならないと同時に、性能、信頼性、再利用性、拡張性等様々な非機能的特性の達成が要求される。本稿ではこれらの特性の中で特に性能に関して議論を行う。

ソフトウェアの性能について考慮する場合、サービスに照らして定義される性能に我々は興味を持つ。例えば、装置を管理するシステムにおいて、障害を検出してから監視側に表示するまでの時間が何秒以内であるかというようなサービスに付随する質としての性能が重要であると考える。したがって、本稿ではこうしたサービスに照らして定義される性能について考えることとする。

こうした性能についての要求を満たすようにソフトウェアを設計することを、性能設計と呼ぶことにする。

2.1 性能の理解の困難さ

ソフトウェアがどのような性能を示すかについては、開発終了段階になって全体を動かしてみるまでは正確に理解することが難しく、現実のソフトウェア開発においても、当初期待していた性能が出せないというような問題が多出している。

設計時に最終的なソフトウェアの性能を理解することが困難である原因として、以下のようなことが挙げられる。

- **因果関係の理解の困難さ**：ソフトウェアの論理構造は、非常に膨大で複雑なものになっている場合があり、全体の性能が何によって決定づけられているのかその因果関係を理解することは非常に難しい。また、性能は実装の細部にも依存し、上位の論理構造とは異なった因果関係を持ち得る。そのため、最終的には一見予測のつかない性能を示すことがある。
- **ブラックボックスの存在**：性能は、そのソフトウェアが動作するハードウェアの性能にも依存する。また、通常、ソフトウェアはOSやミドルウェア等の様々な外部の環境を使用する。これらの性能によっても、ソフトウェア自身の性能は影響を受ける。しかし、ハードウェアを含めたこれら外部の環境は、ソフトウェアにとってブラックボ

ックスであることが多く、その性能を理解することは困難である。

2.2 性能設計の困難さ

前節で述べたように、設計時に性能について正確に理解することは困難であり、実際にソフトウェアを作り動作させて計測してみないとわからないことが多い。

しかし、試験的に一部を実装して性能を測定しただけでは、適切に性能を把握できないことがある。例えば、測定した部分が他の部分とリソースを共有している場合には、一部だけを実装して測定した性能値が最終的にソフトウェアを動作させた場合の値と異なる可能性が高い。また仮にマシン間の通信時間が正確に測定できたとしても、その通信時間と全体のサービスの処理時間との因果関係の理解が困難であるために全体の性能を予測できないこともある。

したがって、性能を正確に捕捉するという目的のためには、ソフトウェア全体を構築してからその性能を実測することが理想的である。しかし現実には、開発の最終段階に全体を動作させられるようになってから性能を確認するのでは問題が発見されても修正をすることが困難であり、妥当な方法ではない。

3 研究の目的

実装・実測を全く行わずに論理構造に基づいて性能設計を行なったり、逆に全体を構築してから動作させて性能を確認するという方法は、現実的には困難である。したがって我々は、一部を実装してその性能を測定し、その知識に基づいて設計する方法が適切であると考える。

しかし2.2で述べたように、一部分の測定値を用いる場合には、以下の点を考慮することが重要である。

- その部分のみを実装し実測した場合の性能が、全体を実装した時に該当部分を実測した場合の性能と大きく変わるようであれば、効果が少ない。
- 仮にその部分の性能が分かったとしても、その知識に基づいて全体の性能を見積もることが難しければ、効果が少ない。

すなわち、実測の対象とする部分の性能と他の部分との関係や、対象部分の性能と全体の性能との関わりなどのソフトウェアの構造を捉え、適切な部分を対象に実装・実測を行わなければ、こうした設計手法は機能しないと考える。

我々の研究の目的は、部分的な実測による全体の性能の見積もりを設計に効果的に利用する手法を見いだすことである。

特に本稿では、性能面からアーキテクチャを捉えることにより、性能を見積もるために適切な実装・実測部分を捉える手法について述べる。

4 性能面のアーキテクチャ

前章で指摘したように、実測に基づいて性能を設計するためには、実測の対象とする部分と他の部分との関係や対象部分と全体の関わりなど、性能面からソフトウェアの構造を捉えることが必要である。

こうした性能面のアーキテクチャを捉えるために、以下の概念を定義する。

- **サービスモジュール群**：サービスSを実現するために必要なモジュール群を、Sに対応するサービスモジュール群と呼ぶ。サービスを実現するために必要な要素を捉えるために、ここではモジュールには一般にリソースと呼ばれるもの、例えばチャネルやキューなども含まれるものとする。
- **協調関係**：ふたつのサービスモジュール群があり、それぞれが対応づけられているサービスを組み合わせて上位のサービスが実現されるとき、両者の間には協調関係があるという。
- **依存関係**：ふたつのサービスモジュール群があり、それぞれが対応づけられているサービスの実行が、お互いのふるまいに影響を及ぼすとき、両者の間には依存関係があるという。

他のモジュール群と協調関係を持っているサービスモジュール群を実測対象とする際には、そのモジュール群の性能が協調した結果実現される上位のサービスの性能とどのような関係を持つかが理解できていなければならない。また、依存関係を持っている部分は、他のサービスの影響で振る舞

いが変わってしまう可能性があるため、そのモジュール群だけの性能を測定しても全体の性能の見積もりが難しいこともある。したがって、性能を理解するためには、このような関係を捉えることが重要である。

5 協調/依存グラフを用いた性能設計手法

実測に基づいて性能を見積る手法において、部分的に実装し実測する対象とする部分を実測対象モジュール群と呼ぶ。性能を見積もるために適切な実測対象モジュール群を判断するとは、前章で述べた協調関係や依存関係に注目して適切なサービスモジュール群の集まりを捉えることである。本章では、前章で述べた性能面のアーキテクチャをグラフで表現することにより、性能設計に有意な実測対象モジュール群を見いだす設計手法について述べる。

5.1 グラフの定義

サービスSの性能に関わる構造を表現する協調/依存グラフとは、以下のノードとノード間の2種の重み付きアークから構成されるグラフである。

- **ノード**：ソフトウェアが実現すべきサービス(サブサービスの場合もある)について、それぞれに対応するサービスモジュール群をノードとする。
- **依存アーク**：サービスモジュール群の間の依存関係をノード間の依存アークとする。
- **依存アークの重み**：ふるまいへの影響が、各々のサービスの性能を変化させる度合を依存アークの重みとする。重みは0以上1以下で定義する。
- **協調アーク**：Sに含まれるサービス(S の下位のサービス)に対応づけられているサービスモジュール群の間の協調関係を協調アークとする。
- **協調アークの重み**：組み合わせるための処理が上位のサービスの性能に対して持つ影響の強さを協調アークの重みとする。重みは0以上1以下で定義する。

図1に協調/依存グラフの例を示す。

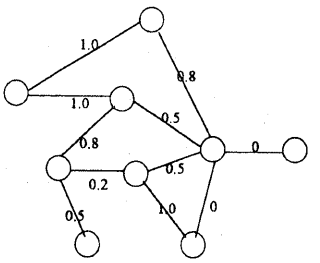

図1　協調/依存グラフ

5.2 グラフの構成法

現実的には、サービスモジュール群の間に潜在的に存在するすべての協調関係や依存関係を発見すること、またその関係によってお互いに及ぼす影響の度合を完全に理解することは難しい。しかし、設計の時点で持っている知識を用いて、近似的な理解をすることは可能である。我々は、完全な協調/依存グラフを作成することを目的としているわけではなく、その時点で持っている知識に基づいて、性能面のアーキテクチャを近似的に表現する協調/依存グラフを作成し設計に利用することが重要であると考える。

より正確な協調/依存グラフを作成するためには、サービスモジュール群を組み合わせて上位のサービスを実現するために実際に行っている組み合わせの処理の詳細や、それぞれのサービスモジュール群の性能を変化させ得る要因、また使用しているリソースの性能の特徴などについての知識が必要である。本節では、我々の経験を踏まえて、こうしたことに関する知識が乏しい場合にも、有効に利用され得るグラフを構築するための構成法を示す。より具体的な知識を持っていれば、それに基づきより正確なグラフを構成することも可能である。

①注目するサービスSを決める。
適切な実測対象モジュール群を捉えることにより性能を見積りたいサービス、すなわちグラフの作成対象となるサービスを決める。

②サービスモジュール群を定義し、ノードとする。
ソフトウェアで実現するすべてのサービス(S 以外も含む)について、それぞれを基本的なサブサービスに分割し、それぞれのサブサービスを実現するために必要なモジュール群をサービスモジュール群とする。それぞれのサービスモジュール群をノードで表す。近年、パターンを活用したモデル化手法が提案されているが[6][8]、そうしたモデル化を利用する場合にはパターンに対応する部分がサービスモジュール群となる。
③サービスモジュール群の間にモジュールの重複があれば、サービスモジュール群の間に依存アークを引く。
ここでのモジュールは、前述のように一般にリソースと呼ばれるようなものも含む。具体的には、チャネル、キューのようなリソースの共有がないかどうか注意する。こうしたモジュールの重複は、お互いのサービスモジュール群の振る舞いに影響を及ぼしうるので、モジュールの重複を依存関係とする。
④依存アークに重みをつける。
サービスモジュール群の間でモジュールを共有している場合に、共有しているモジュールをそれぞれのサービスモジュール群で競合して使おうとすると、そのことにより各サービスモジュール群で実現するサービスの性能は一般に低くなる。このような各サービスモジュール群でのモジュールの競合、同期を取った処理があるかどうかを依存の強さとし、依存アークの重みとする。モジュールの競合がある場合に、重みを 1 とする。
⑤S に含まれる(下位である)サービスに対応づけられているサービスモジュール群の間でモジュールに重複があれば、そのモジュール群を表すノード間に協調アークを引く。
協調動作をする場合、モジュールに重複があると考えられるので、モジュールの重複を協調関係とする。
⑥協調アークに重みをつける。
各サービスモジュール群に対応するサービスを組み合わせて上位のサービスを実現する時に、各サービスモジュール群が想定しているデータモデル、制御モデルに不整合があれば[2]、協調動作する際に必要となる処理が上位サービスの性能に影響を持つと考えられる。具体的には、サービスモジュール群の間でデータ構造の変換やプロトコルの変換が必要な場合である。このような不整合を協調の強さとし、協調アークの重みとする。不整合があれば重みを 1 とする。

5.3 グラフの利用法

協調/依存グラフを利用して、性能の捕捉のために有意な実測対象モジュール群を判定し、性能の部分的な測定を行なう。

実測対象モジュール群を、以下のように判断する。協調/依存グラフ上で、重みがある大きさ以下のアークを切ることにより、独立した 1 個以上の部分を得ることができる。アークの重みはサービスモジュール群の間の依存関係の強さを表しているので、この部分はある強さ以上で結合しているサービスモジュール群の集合であり、他の部分とは弱い関係しか持っていないことになる。したがってこの独立した部分を実測対象モジュール群とすることは、その実測結果が有効に活用できるものであると期待される。

ここで、どの程度の重み以下のアークを切るのかは、グラフにより判定した実装対象モジュール群の実測値をどの程度の精度で使いたいか、また実測にどれくらいのコストをかけられるのか等、開発上の戦略に基づいて決定されるべきである。すなわち、コストがかかってもお互いに影響を及ぼしそうなところは全て取り出して大きな部分を切り出し、より全体の特性に近い値を得ようとするならば、アークを切る重みは小さくなければならない。逆に、影響を及ぼし合うことが濃厚な小さな部分だけを取り出してまずは簡単に確認することが重要ならば、アークを切る重みはかなり大きな値にすべきである。

判定した実測対象モジュール群に対して、それを構成するモジュール群を特定し、その実装と実測を行なう。協調関係により

サービスモジュール群が関係づけられている場合は、それぞれのサービスモジュール群に対応づけれたサービスを組み合わせて実現される上位のサービスについて実装し、そのサービスに関わる性能を実測する。依存関係によりサービスモジュール群が関係づけられている時には、グラフで注目しているサービスに含まれるサービスに対応づけられたサービスモジュール群の性能を実測する。そのモジュール群と依存関係にあるモジュール群については、それ自体の性能を測定することが目的ではないが、同時に動かすことにより注目しているサービスの性能への依存関係を確認する。

注目しているサービスに関わる実測対象モジュール群の性能を実測した後、それらに基づいて要求されるサービスの性能を見積もる。例えば実測したそれぞれの実測対象モジュール群での処理時間を合計して全体の処理時間を見積もるなどの方法が考えられる。

6 例題

本章では、本手法により有意な実測対象モジュール群がどのように発見されるのか例題を用いて簡単に示す。

6.1 例題システム

6.1.1システム概要

装置の監視システムを取り上げる。

このシステムでは、3 種類の装置を監視しており、装置の種類毎に監視するサーバ、それらのサーバを集中管理するサーバ、サーバが管理する装置の状態を表示するビューアからなる。それぞれの装置に異常が起こると、そのことがビューアに表示される。

6.1.2構成と実装の想定

このシステムを実装するにあたって、以下のことが想定されているとする。

- 装置Aから装置A監視サーバへのアラームの通知と装置Bから装置B監視サーバへのアラームの通知はチャネルを用いた実装とする。またこれらのチャネルは同じものを共有する。
- 装置から監視するサーバへの通知には、ミドルウェアで規定されているデータ構造が使われる。
- 集中管理サーバが各監視のサーバの状態を監視するにあたり使われるデータ構造は、ミドルウェアで想定されているデータ構造とは異なるものである。

6.2 グラフの作成と実測対象モジュール群の発見

このシステムにおいて、装置Aに異常が起こったことをビューアに表示するサービスSについて、処理時間を捕捉するために有意な実測対象モジュール群を発見することを考える。

5.2に示した方法で、グラフを構成する。

まず、このシステムを、装置のサービスに注目しサービスモジュール群に分割する。この様子を図2に示す。それぞれのサービスモジュール群をノードとする(図 3)。これらのサービスモジュール群で、モジュールに重複があるのは、P4 と P5、P4 と P6、P5 と P6 であるので、これらの間に依存アークを引く。また、装置Aから装置A監視サーバへのアラームの通知と装置Bから装置B監視サーバへのアラームの通知で同じチャネルが使われるので、P1 と P2 でモジュールの共有が起こっており、リソース共有という依存関係が存在するのでその間に依存アークを引く。S の下位のサービスを実現するサービスモジュール群の間でモジュールの重複があるのは、P1 と P4、P5 と P7 であるので、それぞれの間に協調アークを引く。

各アークの重みづけは、装置からサーバへの通知のサービスモジュール群と、集中監視サーバが各サーバを監視するサービスモジュール群で異なるデータ構造が想定されているため、このサービスモジュール群間の依存アークの重みは、データモデルの不整合により 1 となり、その他の部分には不整合等が発見されないので重みは 0 である。このようにして構築したグラフを図 4 に示す。重みが 1 以下(つまり 0)のアークを切ると、図 5 に示すような独立した部分が発見できる。

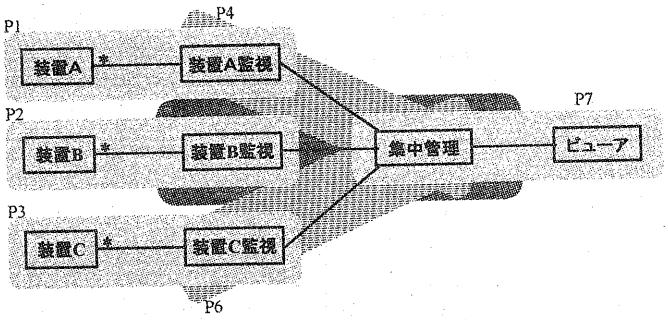

図２　装置監視システム

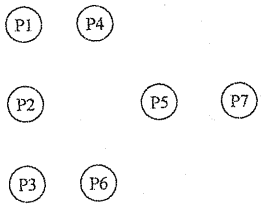

図３　サービスモジュール群

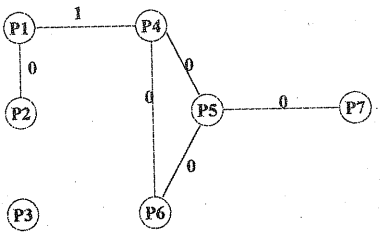

図４　協調/依存グラフ

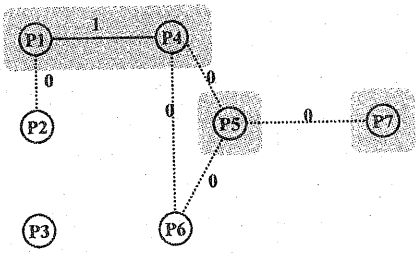

図５　実測対象部分の判定

したがって、装置Ａの異常をビューアに表示するサービスにおいて、装置Ａから装置Ａ監視サーバへの通知のサービスに関わる部分と、集中管理サーバによる装置Ａ監視サーバの監視サービスに関わる部分には依存関係があると考えられるので、それらをひとまとまりとして実測しなければならないと考えられる。また、この部分と集中監視サーバの状態をビューアに表示するサービスに関わる部分には、弱い依存関係しかないと考えられるので、別々に実測できると考えられる。

6.3 知識の増加による実装対象モジュール群の変化

各サービスモジュール群やこのシステムが使用するプラットフォームに関する知識が増えた場合に、発見される実測対象モジュール群がどのように変わっていくか見る。

このシステムで使用しているアラームの通知のためのチャネルが、実はキャパシティが小さく、そのチャネルの利用者が多くなると性能が落ちる可能性があることが分かった。また、この環境では、装置から監視サーバへの通知のサービスモジュール群で使うデータと、集中管理サーバが各装置監視サーバを監視するサービスモジュール群で使っているデータの変換の時間は、無視できるほど小さいことが分かった。

この時、チャネルを共有しているノード間のアークには、データモデルの不整合よりも重い重みを与えることが良いと考えられる。この時の協調/依存グラフグラフとそこから発見される実装対象モジュール群を図６に示す。

この場合のグラフによると、装置Ａから装置Ａ監視サーバへの通知のサービスモジュール群と装置Ｂから装置Ｂ監視サーバへの通知のサービスモジュール群をひとかたまりにして実測することが有意であると考えられる。集中管理サーバによる各装置監視サーバの監視のモジュール群は切り離して実測して良い。

このように、知識の増加により、より精度のよい実測対象モジュール群の発見ができる。

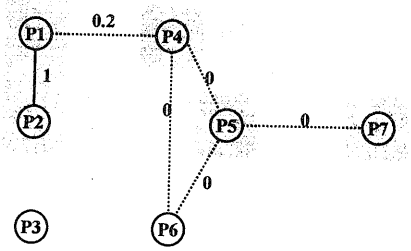

図6　知識の増加による実測対象部分の変化

7　議論

ソフトウェアの性能問題に関しては、性能モデルを定義しそれを基に性能を議論する研究等が行われている[3][4]。また近年、そうした視点とは別に、ソフトウェアアーキテクチャと性能等の非機能的特性の関わりについて考察されるようになっているが[1][5][7]、アーキテクチャの観点から具体的に非機能的特性をどのように設計すれば良いのかは、まだ十分に研究されていない。本研究は、ソフトウェアアーキテクチャという構造面に注目して性能設計をしようとするところに特色がある。

本稿では、サービスモジュール群の間の依存関係の重みづけについて、経験に基づく簡単な重みづけの戦略を示した。この戦略は、我々の経験に基づき、サービスモジュール群や利用するブラックボックスの性能について十分な知識がない場合にも近似的に妥当な重みづけをしようとするものである。しかし、性能の実測の対象となる部分をより高い精度で判断するためには、もっと詳細な重みづけが望まれる。具体的にどのような知識を増やしていけばより有意な重みづけができるのか現状では明らかではなく、さらに詳細な検討が必要である。

本稿では、サービスモジュール群間に依存関係が定義できる場合として、サービスモジュール群の間でリソースを共有する場合を取り上げている。しかし実際の問題を見てみると、1つのCPUを多くのプロセスで使おうとして性能が落ちる等、全体においてリソースを共有しているものの数が性能に影響を及ぼす状況がよくある。このようなリソースのグローバルな共有が性能に与える影響を捉えるために協調/依存グラフを活用するのが妥当であるかどうかは、今後の検討課題である。

8　おわりに

本稿では、実装、実測した結果をソフトウェア全体の性能を捕捉するために効果的に活用できると期待される部分を、協調/依存グラフを利用して特定する手法を提案した。

今後は、実際の開発にこの手法を適用して有効性を確認したい。

参考文献

[1] Buschmann, F., et.al.: *Pattern-Oriented Software Architecture - A System of Patterns,* Wiley, (1996).

[2] Garlan, D., et.al.: *Architectural Mismatch: Why Reuse is So Hard,* IEEE Software, Nov. (1995)

[3] Gomaa, H.; *Software Design Method for Concurrent and Real-time Systems,* Addison-Wesley, (1993)

[4] Hatley, D.J., et.al., *Strategies for Real-Time System Specification,* Dorset House Publishing, (1988). 邦訳: リアルタイム・システムの構造化分析, 日経BP.

[5] Krucheten, P.B.: *The 4+1 View Model of Architecture,* IEEE Software, Nov. (1995)

[6] 野田,他: パターンを用いたアーキテクチャ設計, 情報処理学会, OO'97, (1997).

[7] Soni, D., et.al.: *Software Architecture in Industrial Applications,* Proceedings of the 17th International Conference on Software Engineering, April (1995).

[8] 山本, 他: 金融アプリケーション開発におけるパターン適用, 情報処理学会, OO'97, (1997).