ソフトウェア工学 120－17
（1998．7．24）

アーキテクチャコンフォーマンスに基づくソフトウェア設計

岸 知二, 野田夏子

NECソフトウェアデザイン研究所

ソフトウェア開発においては一義的に求められる機能的特性だけでなく、性能や信頼性といった非機能的特性の達成も重要な課題である。しかしながら非機能的特性をどのように設計するかについて体系だった手法はなく、実際の開発では性能問題等が頻出している。本稿ではアーキテクチャという粗粒度のソフトウェア構造に注目し、部分的に確認された非機能的特性の知識に基づいて、全体の非機能的特性をより確度高く実現する手法について提案する。そのために非機能的特性からみたソフトウェアアーキテクチャと、それへのコンフォーマンスの概念を導入する。

Software Design based on Architecture Conformance

Tomoji Kishi, Natsuko Noda

Software Design Laboratories
NEC Corporation

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 examining 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, the notion of architectural conformance, and propose the design method based on the conformance.

1 はじめに

ソフトウェア開発においては一義的に求められる機能的特性だけでなく、性能や信頼性といった非機能的特性の達成も重要な課題である。しかしながら、設計において非機能的特性を考慮し達成するための手法は十分に確立されておらず、現実の開発においては期待した性能が出ないといった非機能的側面に関る問題が日常的に発生している。

我々はソフトウェアアーキテクチャ[3][7][9]の視点から、こうした非機能的特性をより確実に設計する手法を検討している[8]。この手法では、アーキテクチャという粗粒度のソフトウェア構造に注目し、部分的に確認された非機能的特性の知識に基づいて、全体の非機能的特性をより確度高く実現することを狙っている。その目的のために、非機能的特性が確認された部分的なモジュール群を利用して構築されるソフトウェアのアーキテクチャに対する制約と、その制約に対するアーキテクチャコンフォーマンスの概念を導入する。

2 章では非機能的特性の設計に関する問題を指摘する。3 章では我々の研究の目的を述べる。4 章では非機能的特性からのソフトウェアアーキテクチャと、そのコンフォーマンスの概念を導入する。5 章ではコンフォーマンスに基づく設計手法について述べ、6 章で例題を示す。

2 非機能的特性の設計

本章では非機能的特性の定義と、設計時における問題点について述べる。

2.1 非機能的特性とは

ソフトウェアは一義的には要求される機能を実現することが求められるが、そうした機能的特性では捉えられない特性、例えば性能、信頼性などを非機能的特性と呼ぶ[2]。再利用性や拡張性といった特性も非機能的特性と呼ばれるが、本稿ではこうした開発や運用に関る特性ではなく、性能のような実行時の特性のみを対象とする。また求められる非機能的特性を達成するようにソフトウェアを設計することを、非機能的特性の設計と呼ぶ。

2.2 非機能的特性設計上の問題

ソフトウェアの非機能的特性を決定する要因を理解することは一般に困難である。またソフトウェア開発では部品やプラットフォーム等、ブラックボックスとして扱わざるを得ないものも多く、それが理解を一層困難にしている。従って実際の開発では重要と思われる部分の非機能的特性を実測し、その知識に基づいて設計を行うことがよく行われている。

しかしながら、こうした手法が適切に機能しないことも多い。例えば図 1 はあるエラー検知の機能の特性を実測したデータであるが、エラー検知の応答時間が、エラーの監視周期やその時のシステム全体の CPU 負荷に応じて大きく変化していることが分かる。このように利用局面によって、あるいは単体で動作させた場合と最終的なシステム全体の中で動作させた場合とによって大きく性能が異なってしまうため、部分的な実測に基づいて設計をする際には注意が必要となる。

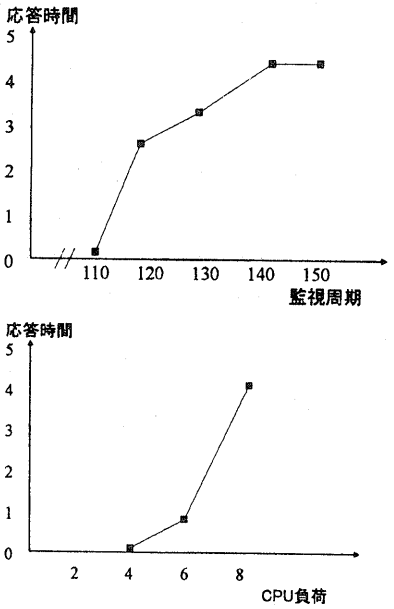

図 1 非機能的特性の実測例

実測に基づいた設計アプローチは基本的には現実的な方法であると考えるが、それをより効果的に機能させるためには、設計の進め方等に関するガイドラインを明確にすることが必要である。

3 研究の目的

我々の研究の目的は、実測等によって得られた非機能的特性に関する知識に基づいて、できるだけ間違いの少ない非機能的特性の設計を行う手法を見いだすことである。特に本稿では、実測アプローチを行う際に考えなければならないアーキテクチャ上の制約を定義するとともに、その制約に対するコンフォーマンスに基づいた設計手法を提案する。

本手法は、最終的な非機能的特性を正確に見積もったり、絶対に間違いのない設計を行うためのものではなく、その時点の知識に基づいてリスクが少ないと期待される設計の指針を与えることを狙ったものである。

4 非機能的特性とアーキテクチャ

本章では、非機能的特性の観点から見たアーキテクチャについて述べる。

4.1 サービスモジュール群

ソフトウェア開発において関心のある非機能的特性は、そのソフトウェアが提供すべきサービスに関る非機能的特性である。ソフトウェアは複数のモジュールから構成されているが、本稿ではその中で特定のサービスの実現に機能的に関るモジュール群をサービスモジュール群と呼ぶ。

あるサービスに関する非機能的特性は、必ずしもそのサービスに対するサービスモジュール群だけで決定づけられる訳ではない。例えばそのサービスの実現で用いている通信チャネルが、他のサービスの実現でも用いられていた場合、その競合によって非機能的特性が変化することがある。そうした場合には、他のサービスのサービスモジュール群もそのサービスの非機能的特性に影響を与えていることになる。

従ってあるサービスの非機能的特性を実測する際には、対応するサービスモジュール群だけを実装し実測するだけでは適切な知識が得られないかもしれない。しかしながらどういう要因が非機能的特性を決定づけているかという十分な知識がないのであるから、どの範囲を実装し、どのような実測を行うかはその時点の設計者の判断にゆだねられることになる。すなわち設計時に利用できる非機能的特性の知識は、上記の判断によって決定されたモジュール群(この中には関心のあるサービスに対応したサービスモジュール群だけでなく、他のサービスに関るサービスモジュール群も含まれ得る)に対する知識ということになる。

4.2 非機能的特性のアーキテクチャ

ソフトウェアを設計する際には、こうした知識を踏まえ、全体として適切な非機能的特性を達成するように設計を行う必要がある。

一般にソフトウェアは複数のサービスを提供し、そのサービスの実現のために内部的には様々なサブサービスを実現する必要がある。これらのサービスやサブサービスに対応するサービスモジュール群と、それらの間の非機能的特性に関する関係をアーキテクチャとして捉えることを考える。なお前節で述べたように、あるサービスの非機能的特性は、そのサービスに対応したサービスモジュール群だけで決定されるわけではない。本アーキテクチャで議論するサービスモジュール群間の関係はそれらを含めたうえでの関係を表現しているものとする。

サービスモジュール群Sを実現するにあたり複数のサービスモジュール群SLi(iはSが利用するサービスモジュール群の数までの自然数)を利用し、またSは複数のサービスモジュール群SUj(jはSを利用するサービスモジュール群の数までの自然数)の実現に利用されているとしたとき、サービスモジュール群Sが規定する非機能的特性に関するアーキテクチャA(S)を以下のように定義する。

A(S) = {
(a1) サービスの提供に関して SUj に課せられる構造的制約、

(a2) 非機能的特性の目標値、
(a3) (a2)を達成するために SUj に課せられる制約、
(a4) 各 SLi に課せられる非機能的特性の目標値
}

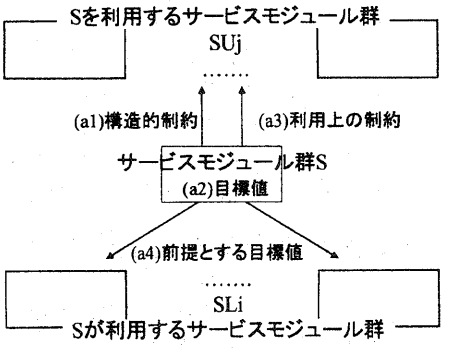

図 2 規定されるアーキテクチャ

(a1)は S がサービスを提供する際に、そのサービスを利用する他のサービスモジュール群との間に想定している構造的な関係を示している。サービスは対応するサービスモジュール群が外部と協調しながら提供されるが、その協調関係の想定が異なると非機能的特性は異なってしまう。例えばあるデータにアクセスする際に、特定の関数群を仲介することを想定しているのに、それをバイパスして利用されると、データアクセスというサービスは実現されても、非機能的特性は変わり得る。非機能的特性を設計するには想定する使われ方を明確にする必要がある。

(a2)はこのサービスモジュール群に対して設定される非機能的特性の目標値である。なお複数のサービスモジュール群が S を利用するとしても、ひとつの S に対する目標値はその設計の中で整合していなければならない。

(a3)は(a2)を達成するために SUj に対して課す制約条件である。例えば通信速度が通信するデータ量によって変化するならば、特定の性能を期待するためにはデータ量に対して制約がつけられるかもしれない。こうした制約を意味している。

(a4)は S が利用する各サービスモジュール群に対して S が設定した非機能的特性の目標値である。S は利用するサービスモジュール群がこの目標値を達成していることを前提にして、(a2)の目標を実現する。

以上の関係を図 2 に示す。

4.3 アーキテクチャコンフォーマンス

非機能的特性の設計とは、各サービスモジュール群が規定するアーキテクチャが相互に整合する形に全体を体系づけることであると捉えることができる。そこで上記のアーキテクチャに対するコンフォーマンスの概念を導入する。本アーキテクチャの制約はサービスモジュール群が相互に与えあうものであるため、このコンフォーマンスは、各サービスモジュール群にとっての SLi や SUj から課せられる制約に対して定義される。

サービスモジュール群 S のアーキテクチャコンフォーマンス C(S)を以下のように定義する。

C(S) = {
(c1) 各 SLi が課す(a1)への適合、
(c2) 各 SLi が課す(a3)への適合、
(c3) (a2)と、各 SUj が課す(a4)との整合
}

(c1)は S が SLi を利用する際に、SLi 側が想定している使い方を、S が守っているかどうかを示す。

(c2)は SLi を目標値どおりの特性で利用するために SLi 側が利用する側に課す制約を、S が守っているかどうかを示す。

(c3)は S の目標値が、SUj が想定している目標値と整合しているかどうかを示す。

5 コンフォーマンスを活用した設計手法

本章では、非機能的な側面に関するアーキテクチャコンフォーマンスの考え方を、実際の設計に適用する方法について述べる。

5.1 レイヤシステム

前章で述べた非機能的側面に関するアーキテクチャは、構築されるソフトウェアにおいて最終的に成立すべき制約関係について言及しているが、こうした制約関係を満たしたソフトウェアの構造をどのように実現すべきか、そのガイドラインが必要となる。

通常の開発においては機能的な側面からアーキテクチャを構築することが多いため、そうした構築の枠組みの中で、本アーキテクチャコンフォーマンスの考え方をどのように活用するかを検討する。本稿では、そのひとつの典型的な形態として、レイヤシステムに基づいたソフトウェアの実現を想定し、その手法を述べる。レイヤシステムは階層的に構成され、各レイヤは上位のレイヤに対してサービスを提供し、下位のレイヤに対してのクライアントとなる[11]。

あるレイヤが提供すべきサービス群に対応する複数のサービスモジュール群から、そのサービスの実現に用いられた下位のレイヤのサービス群に対応する複数のサービスモジュール群を除外した部分が、そのレイヤに含まれることになるため、それがそのレイヤにおける設計の対象となる。

前章で述べたサービスモジュール群のアーキテクチャに基づいて、レイヤ間(レイヤに含まれるモジュール群間)のアーキテクチャを定義することができる。

すなわちあるレイヤが提供するサービス群のそれぞれに対応するサービスモジュール群が上位のサービスモジュール群に対して課す制約(a1)(a3)の集合が、そのレイヤが上位のレイヤに課す制約となり、サービス群のそれぞれに対して設定されている目標値(a2)の集合がそのレイヤの目標値となる。またそのレイヤが利用する下位のレイヤの提供するサービス群それぞれに対応するサービスモジュール群に対して課せられる目標値(a4)の集合が、そのレイヤが下位のレイヤに対して課す制約となる。

一方、あるレイヤが下位のレイヤのサービス群を利用して実現されている場合には、利用するそれぞれのサービスに対応づけられた制約が課せられることになり、その制約を満たすことがコンフォーマンスとなる。

構造的制約を例に取ると、そのレイヤが提供するサービスごとに、それを利用する際に課せられる構造的制約群が定義され、そのサービスを利用する上位のレイヤ中のモジュールがその構造的制約を守ることが、それへのコンフォーマンスとなる。

5.2 非機能的特性の設計

最上位のレイヤが提供するサービスは、そのシステムが提供するサービスそのものであり、そのサービスは下位のレイヤの提供するサービスによって実現されるものとする。このとき、これらのサービスに関る非機能的特性をどのように達成するかその手法について考える。

本手法は、非機能的特性に関する要求事項のブレークダウンと、確認された知識に基づくインクリメンタルな実現という2つの作業から構成される。

システムの提供するサービスに対して非機能的特性に関する達成目標が与えられたとき、それが最上位のレイヤに対して課せられる目標値となる。あるサービスに対して非機能的特性の目標値が設定されたとき、そのサービスを実現するために利用している下位のレイヤのサービス群に対して、それぞれ非機能的特性の目標値がブレークダウンされる。これは(a2)に基づいて(a4)を決める作業に相当する。

ここでブレークダウンとは、そのサービスの非機能的特性の達成において、それが利用するサブサービスの非機能的特性に対して目標値を設定することを意味する。例えばサービス全体の応答時間を各サブサービスに配分することもありうるし、サービス全体に求められる信頼性と同様の信頼性を、各サブサービスに期待することもありうる。

一方インクリメンタルな実現とは、ブレークダウンされた非機能的特性の要求を踏まえたサービスの実現を決定し、それに基づいてそのレイヤ全体を設計することである。この際に、下位のレイヤの非機能的特性についての知識を活用する。すなわち、

下位のレイヤに対して期待する目標値(a4)を達成するために下位レイヤから課せられる制約(a3)が定義されている合には、それへのコンフォーマンスを確認しながらそのレイヤを設計することになる。

5.3 コンフォーマンスの確認

本節では実際にどのようにコンフォーマンスを確認するのか、その確認方法について述べる。ただしどのような非機能的特性を対象としているのか、具体的にどのような実装に基づくサービスを議論しているのか等によって制約事項の内容が異なるため、一般的な確認方法を議論することは困難である。ここでは典型的な状況を想定し、一部例示的な形での提示を行う。

なお確認の対象となるのは設計であり、典型的には設計モデルのような形態で記述されているものとする。以下、コンフォーマンス定義の各項目の確認方法について述べる。

(c1)そのレイヤが利用している下位のレイヤのサービスそれぞれに対して、構造的制約(a1)に対するコンフォーマンスを確認する。一般に構造的制約には静的側面と動的側面があり、例えばデザインパターン[4]的な定義方法が考えられる。静的側面については、構造記述されたコンポーネントの役割とレイヤ中のモジュールとを対応づけることができるかどうかを確認し、動的側面については、設計された協調関係が、構造記述されたイベントの半順序に関する制約に適合することを確認することによって、コンフォーマンスを確認することができる。こうした確認をモデルのシミュレーションなどで支援することも考えられる。

(c2)そのレイヤが利用している下位のレイヤのサービスそれぞれに対して目標値を達成するための制約(a3)に関するコンフォーマンスを確認する。制約の一般形を議論することは難しいが、典型的な制約として以下の2種類が考えられる。

ひとつはそのサービスの利用方法に直接関る制約であり、通信する際のデータサイズ、エラー監視を行う際の周期等が該当する。これらはレイヤ中でそのサービスを利用している部分の設計を確認することによって判断できる制約事項である。

もうひとつは複数のサービスで共有されるリソースに関る制約であり、ひとつの通信チャネルのキャパシティや、CPU全体での負荷などが該当する。これらはそのサービスに関るモジュールだけでなく、そのレイヤ中でそのリソースを共有しているすべてのモジュールを横断的に確認することによって判断される制約事項である。

なおすべての制約事項が設計上で容易に確認できるとは限らないが、これらの制約事項はその時点の知識に基づいて設計者が設計の指針として定義する制約事項であり、そもそも確認不可能な制約を定義することは設計行為として意味が乏しい。

(c3)設定された目標値が上位のレイヤが期待する目標値を満たしていることを確認する。応答時間の目標値が利用側の期待値を満たしているかを確認するなどの作業に相当する。

6 例題

以上の設計手法を、我々が実際に開発したソフトウェアのエラーを監視するサービス(ウォッチドッグタイマ)に照らして例示すると以下のようになる。

アプリケーションが提供するサービスS1を信頼性高く作るためにエラー監視機能を組み込む状況を考える。S1の実現にはサブサービスとしてS11(本来のサービスを実現するために必要なサービスとする)とエラー監視サービスS12が利用される。またエラー監視サービスS12の実現には、リアルタイムOS(RTOS)の提供するタスクサービスS121と割り込みサービスS122が利用されるとする。なおRTOSは既存のプラットフォームとし、設計の対象ではないものとする。

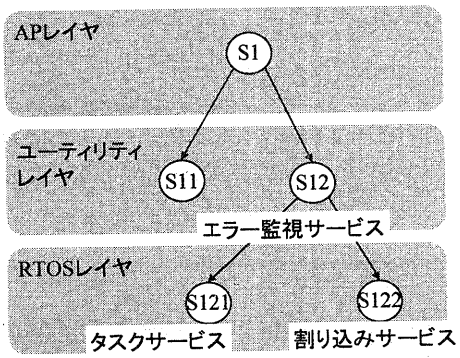

図 3 サービスモジュール群とレイヤの例

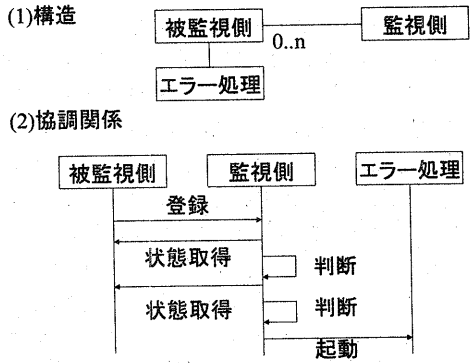

図 4 構造的制約の例

図 3 で丸はラベルで示されるサービス s を実現するために、該当するレイヤで実現されるモジュール群を表わし、以下 m(s)と示す。またサービス s のサービスモジュール群を M(s)と示す。M(S1)には m(S1)、m(S11)、m(S12)、m(S121)、m(S122)が含まれ、M(S12)には m(S12)、m(S121)、m(S122)が含まれる。各レイヤは一般に複数のサービス群を提供しており、例えば AP レイヤを設計する際の設計対象モジュールは m(S1)、ユーティリティレイヤを設計する際の設計対象モジュールは m(S11)と m(S12)である。

このとき S12(エラー監視サービス)に対応するサービスモジュール群 M(S12)が規定するアーキテクチャは、例えば以下のように定義される。

(a1)エラー監視サービスを利用する際の構造ならびに協調関係に対する制約を定義する。図 4 にデザインパターン的な記述でその制約を示す。

(a2)例えば異常検知の応答時間に対する目標値(0.1 秒以下等)が与えられる。

(a3)期待した応答時間を達成するための制約を与える。例えば図 1 で示すように、監視の周期や全体の CPU 負荷によって応答時間が大きく変動するなどの知識が得られた場合、それに基づいて例えば監視されるサービスのタスク周期や全体の CPU 負荷の上限値などが制約として定義される。

(a4)利用する RTOS に対して、タスクや割り込みのスイッチ時間などに関する目標値が課せられる。プラットフォームそのものは設計対象ではないが、その目標値に見合ったプラットフォームを選択する、あるいは利用可能なプラットフォームが達成可能な目標値を設定する必要がある。プラットフォームが異なれば期待可能な目標値も変わり、上位の設計や達成できる最終的な非機能的特性に影響が出る。

同様に、各サービス毎にそのサービスモジュール群の規定するアーキテクチャを定義することができる。

次に上述したようなサービスモジュール群の規定するアーキテクチャに基づいてレイヤ間のアーキテクチャを考える。S12(エラー監視サービス)に関して、ユーティリティレイヤが AP レイヤに対して課す制約は上記の(a1)と(a3)、その非機能的特性の目標値は(a2)、また RTOS レイヤに課す制約は(a4)になる。

一方 AP レイヤからユーティリティレイヤに対しては、M(S1)が規定するアーキテクチャを考えることにより、M(S11)や M(S12)に対する目標値がそのアーキテクチャにおける(a4)として定義される。

なお AP レイヤは最上位レイヤなので上位レイヤへの制約は不要であり、また RTOS レイヤは最下位レイヤなので、下位レイヤに対する制約は不要となる。

ユーティリティレイヤを設計する際には、RTOS から課せられる制約(M(S121)、M(S122)等が規定する(a1)や(a3))を守ってそのレイヤ中のモジュールが設計されてい

るかどうか、RTOS レイヤに設定された目標値(M(S121)、M(S122)等が規定する(a2))とユーティリティレイヤが期待する目標値(M(S12)等が規定する(a4))とが整合しているかどうかを確認する必要がある。同様にAP レイヤを設計する際には、ユーティリティレイヤから課せられる制約(M(S11)、M(S12)等が規定する(a1)や(a3))を守ってそのレイヤ中のモジュールが設計されているかどうか、ユーティリティレイヤに設定された目標値(M(S11)、M(S12)等が規定する(a2))が AP レイヤが設定している期待値(M(S1)等が規定する(a4))と整合しているかどうかを確認する。

7 議論

非機能的特性に関する研究はリアルタイムシステムなどにおける性能のモデリング等様々な研究がなされている[5]。本研究はそうしたアプローチとは異なり、開発段階で持ち合わせているその時点までの知識に基づき、妥当と考えられる設計の指針を得ようと言うものである。こうした粗粒度の構造の認識は、厳密なモデル化と違って正確な見積り等に利用できるものではないが、逆に大局的な問題構造を認識する上で重要であり、既存のアプローチと補完的な視点であると考えている。

なおアーキテクチャ的な観点から特性を検討する手法も提案されているが [1][6]、本手法は実装レベルの制約関係を捉え、アーキテクチャコンフォーマンスを活用するなどの点でアプローチが異なる。

本稿で指摘したように、サービスモジュール群を識別して部分的に実装、実測しても、場合によっては全体の非機能的特性の見積り等に効果的に活用できないこともある。どのような範囲を実装、実測の対象として捉えるかは重要な問題であり、我々はそれに対する検討も進めている[10]。

なお(a3)等の制約をどう定義するか、複数の制約が矛盾する場合にはどうなるか等、制約の定義や扱いについては多くの課題が残されており、今後いっそう検討が必要である。

8 おわりに

以上、本稿ではアーキテクチャコンフォーマンスに基づく非機能的特性の設計手法について検討した。本手法は過去の開発事例の経験を一般化したものであり、今後より検討を深め実用性を高めていきたい。

参考文献

[1] Buhr, R.J., et.al.: *Use CASE Maps for Object-Oriented Systems*, Prentice-Hall, (1996).

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

[3] Clements, P.C., et.al: *Software Architecture: An Executive Overview*, Component-Based Software Engineering - Selected Papars from the SEI, IEEE, (1996).

[4] Gamma, E., et.al.: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, (1995).

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

[6] 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.

[7] 岸知二: ソフトウェアアーキテクチャモデルに基づく設計手法について, FOSE'96, (1996).

[8] 岸知二：アーキテクチャパターンに基づく設計手法の考察、SIGSE, Vol97, No.74, (1997).

[9] 岸知二：ソフトウェアアーキテクチャへの招待、FOSE'97, (1997).

[10]野田夏子、岸知二: 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、SIGSE, Vol.98, No.39,(1998).

[11]Shaw, M., et.al.: *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, (1996).