ソフトウェア工学 112－7
（1996．11．15）

ソフトウェアアーキテクチャに基づく安全性ソフトウェアの開発

岸 知二*、川口 晃*、駒嵜克郎†

*NECマイコンソフト開発環境研究所
†JR東日本 総合技術開発推進部

安全性に関わるシステムにおいてもソフトウェアの比重がいっそう高まる傾向にあるが、ソフトウェアが複雑化するにつれ、安全性を議論できるドメインエキスパートがその設計の妥当性を判断することが困難になってきている。またソフトウェアの開発量が増加し、関わるソフトウェア開発者数が増えることにより、設計品質のばらつきも問題化してきている。我々は安全性ソフトウェアのアーキテクチャを定め、それに基づいたプラットフォームを整備することにより、そうした問題を改善するアプローチを検討した。本稿では、安全性ソフトウェアのソフトウェアアーキテクチャ、それに基づいた安全性ソフトウェアのためのプラットフォーム、さらにそれらを利用した開発プロセスについて、鉄道信号用ソフトウェアに対して試験適用した事例を報告する。

Development of Safety Software based on Software Architecture

Tomoji Kishi*, Akira Kawaguchi*, Katsuro Komazaki †

*Microcomputer Software Development Laboratory
NEC Corporation
† Technical Development & Research Department
East Japan Railway Company

Software safety is crucial because the importance of software in safety systems becomes large. In order to improve the quality of safety software, we have been examining the design method and platform for safety software. In this paper, we present software architecture for safety software, and design method based on the architecture.

1 はじめに

安全性に関わるシステムにおいてもソフトウェアへの依存度は年々高まっている。従ってそうしたシステムに用いられるソフトウェア(以下安全性ソフトウェアと呼ぶ)が安全性を脅かすことのないようにすることがいっそう重要となってきている。またソフトウェアは物理的な法則に従うハードウェアとは異なる特性を持っているため、安全性達成のためにはソフトウェア固有の安全性議論も必要である。我々は安全性ソフトウェアのソフトウェアアーキテクチャを定型化し、それに基づいた安全性ソフトウェアのプラットフォームと開発手法を検討してきた。本稿ではソフトウェアアーキテクチャという観点から本検討について報告する。

一般にソフトウェアを実現する際には、要求されるサービスの実現という機能的な側面だけでなく、修正・拡張容易性、性能、信頼性、安全性などの非機能的な側面への考慮が重要となる。デザインパターン[2]に代表されるような従来のアーキテクチャ検討は、修正・拡張容易性に主体をおいたものが多い。フレームワークの設計においても、ホットスポットに注目することにより、想定されるサービスや実現手段のバリエーションへ対応できる構造を見つける手法などが提案されている[9]。しかしながらフォールトトレラント機能のような安全性機能はそのホットスポット部分についても間違いなく組み込まれる必要があるため、安全性の観点から考えると、こうしたソフトウェアアーキテクチャは必ずしも適切とはいえない。我々は安全性機能を理解容易にするとともに、開発者のスキルにできるだけ依存しない形で実現できるように、安全性実現のためのメカニズムの制御面に注目したソフトウェアアーキテクチャを設計し、それに基づいて安全性ソフトウェアを実現する手法を検討した。

第2章では、安全性ソフトウェアのためのソフトウェアアーキテクチャを検討した背景について述べる。第3章では、本ソフトウェアアーキテクチャと、それに基づく安全性ソフトウェアのためのプラットフォームについて述べる。第4章では、本ソフトウェアアーキテクチャに基づく安全性ソフトウェアの開発手法について述べる。第5章では、本ソフトウェアアーキテクチャや開発手法の妥当性について考察を行う。第6章では関連研究について触れる。

2 背景

本章では、本研究の背景について述べる。

2.1 安全性ソフトウェア

今回検討の対象とした安全性ソフトウェアは、鉄道信号用ソフトウェアである。その典型例としては、連動装置に用いられるソフトウェアが挙げられる。連動装置は鉄道の信号機や転轍機などを操作者の指示や列車の動きに即して間違いなく制御するための装置であり、高度な安全性の要求されるシステムである。こうしたシステムにおいてもソフトウェアが重要な役割を果たしており、その安全性が重要な問題となっている。

こうしたソフトウェアの特性として、以下を指摘することができる。

- 外界からのイベントに応じてリアルタイムに装置を制御する。
- 装置の制御方法に関して多様かつ複雑なルールや制約が存在し、それらに基づいた制御をする必要がある。
- 制御の論理が線形に依存するため、駅ごとに安全性の確認が必要である。

2.2 課題

一般に安全性は信頼性とは異なり、実際にシステムのおかれた現実世界に引き起こされる事象の重要性に照らして議論される必要がある。したがって対象となる現実世界に関しての深い理解を持つ人(以下ドメインエキスパートと呼ぶ)でなければ安全性に関する適切な判断は不可能である。例えば連動装置においても、個々のルールや制約の背景について正しく理解をせずにソフトウェアの開発をすることは極めて危険である。あるいはエラー発生時に全停止にすべきか、部分停止にすべきか等の判断は、安全性と経済性とのバランス判断が必要であり、ソフトウェア開発者が勝手に判断すべき問題ではない。

しかしながらソフトウェアが複雑化、大規模化する中で、ドメインエキスパートがソフ

トウェアのメカニズムを理解し、それが安全に設計されているかどうかを判断することが困難になってきている。一方開発量の増大、開発期間の短期化という状況の中で、安全性を踏まえたソフトウェア設計ができる開発者を十分に確保することが現実的には困難になっており、結果としての設計品質に属人的なばらつきが出てくるという問題も生じている。

システム機能の肥大化、関わる技術の複雑化の中で、安全性ソフトウェアの開発においても開発者の役割分化は避けられない趨勢であり、そうした状況に対応できる安全性ソフトウェア開発の枠組み議論が重要性を増してきている。

2.3 改善のアプローチ

我々は前項で指摘した課題を改善するために、安全性ソフトウェアのアーキテクチャを定め、それに基づいたプラットフォームや開発手法を整備するアプローチを検討してきた。その基本的なねらいは、ソフトウェアの専門家ではないドメインエキスパートが、安全性の観点からそのメカニズムの妥当性を議論しやすくすることと、安全性設計の属人的なばらつきを少なくすることの二点である。

物理的な法則に支配されるハードウェアと異なり、ソフトウェアは組み込まれたロジックに依存したふるまいをするため、そのロジックそのものを理解できないと、安全性設計の妥当性を判断できない。例えば、断線したリレーの接点は重力の力で特定の側に落ちることが高い確率で期待されるため、そちらが安全側になるように設計されているかどうかを確認するという確認ノウハウを持つことができる。しかしながらソフトウェアにおいては、そうしたふるまい上の法則がほとんど存在せず、個別のロジックに依存するため、ドメインエキスパートはロジックの詳細に立ち入らないと、妥当な設計になっているかどうかを判断できない。今回のねらいの第一は、ソフトウェアの機能単位に対して、それに関わるエラーの検知やエラー検出時の処理のしくみをパターン化できるように、制御構造に大きな枠組みをはめることにある。そうすることにより、ドメインエキスパートはそのパターンに基づいて、安全に設計されているかどうかを確認しやすくなる。

一方ソフトウェアの実現方法は多様であるが、それを間違いなく実装するためにはそれなりの考慮が必要となる。例えばエラー処理を優先度高く実行するために特定のタスクの優先度を高くしたとしても、他のタスクの優先度とのかねあいや、タスクの処理内容、タスクの数などが妥当なものでなければ、そのタスクを確実に実行できる保証はない。また現実にはターゲット環境の特性も考慮する必要があり、適切な設計をするためには多面的な検討や知識が必要となる。さらに設計の質は開発者のスキルに依存するため、間違った設計が入り込む危険性もはらんでいる。今回のねらいの第二は、プラットフォームや手法を整備することによって妥当性の確認された方式に沿った設計を方向づけ、こうした開発時の問題を改善することである。

3 安全性のためのアーキテクチャ

本章では、安全性ソフトウェアのアーキテクチャについて、そのねらい、設計、それに基づいたプラットフォームについて述べる。

3.1 ねらい

前章で述べたように本アーキテクチャの目的の第一は、ドメインエキスパートにとって設計の安全性を理解しやすくするために、安全性機能のふるまいをパターン化することにある。ここで安全性機能とは、ハザードを除去、削減、減少させるための機能をいう。

ここで典型的なフォールトトレラント機構として、エラー[1]を検知しエラー処理を実行することを考える。一般にエラー処理として何を実行すべきかは、その時点で実行している機能やシステムのおかれた状況などに照らして判断されるため、エラーが検知された時点での処理の内容や状況が理解できないと、エラー処理の妥当性が判断できない。

従来はまず機能の実現方法を決定し、その実現方法に依存してエラーを検知し、そのた

[1] ソフトウェアの場合何がフォールトであるかの厳密な把握が困難である。本稿ではフォールトとエラーの区別を明確にしていない。

めのエラー処理を個々に設計していたため、その妥当性の判断のためには、機能の実現方法の理解が必要であった。

今回の我々のアプローチでは、まず安全性機能の実現アーキテクチャを決定する。すなわちエラーをカテゴリに分類し、それぞれのカテゴリごとのエラー検知やエラー処理の枠組みを固定化してしまう。そうすることにより、ドメインエキスパートは各機能に照らして、どのカテゴリのエラーが起こった場合にどのようなエラー処理をすればよいのか、という点のみを考えればよく、機能の実現の詳細に立ち入る必要がなくなる。なおソフトウェア開発者は、その安全性機能の実現アーキテクチャの枠の中で、個々の機能の実現方法を検討することになる。

ソフトウェアアーキテクチャは、全体を支配するグローバルなアーキテクチャと、その枠組みの中で局所的に用いられるマイクロアーキテクチャとに区別することができる[5]。そのソフトウェアが要求されている基本的なサービスの実現に必要なアーキテクチャと、安全性機能を実現するために必要なアーキテクチャのどちらをグローバルアーキテクチャと位置づけるかによって、ソフトウェアの構造も開発手法も全く変わってくる。我々のアプローチは安全性機能を実現するためのアーキテクチャをグローバルアーキテクチャ、サービス機能を実現するためのアーキテクチャをマイクロアーキテクチャとして位置づけるという、通常とは逆の考え方を取るものである。

3.2 アーキテクチャの設計

今回のソフトウェアアーキテクチャは、3種類のアーキテクチャによって階層的に実現されている。

3.2.1基本制御アーキテクチャ

基本制御アーキテクチャとは、安全性機能を動作させるための基本的な制御機構であり、具体的には優先度の高い処理を確実に実行させるという、最も基本的かつ根幹に関わるアーキテクチャである(図1)。

こうしたアーキテクチャは、抽象的なレベルで捉えると自明な印象を受けるが、現実にそれを間違いなく実装するためには、注意深い設計が要求される。しかもこうした機構は局所的な設計からは導くことができない。例えば複数のタスクが実行されている中で、特定のタスクが間違いなく実行されるための機構は、すべてのタスクの優先度がどう設定されているかに依存する。安全性機能やサービス機能を実現する際にも、この全体的な枠組みを崩さないように配慮しなければ、基本的な制御構造は保証されない。こうした考慮から基本制御アーキテクチャを最も支配的なグローバルアーキテクチャとして位置づけた。今回は、4段階の優先度を設定し、ターゲット環境で提供されるタスクの優先度、スケジューリング、呼び出し方法などを利用してその実現方法を決定した。

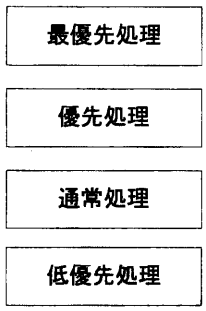

図1：基本制御アーキテクチャ

3.2.2安全性機能アーキテクチャ

次に重要なアーキテクチャとして、安全性機能を実現するためのアーキテクチャを位置づけた。これはエラーをカテゴリに分類し、カテゴリごとにエラー検知やエラー処理の枠組みを規定するものである。この安全性機能アーキテクチャは基本制御アーキテクチャの枠組み(制約)の下で実装される。

今回対象とした安全性ソフトウェアのドメインに照らして、エラーを以下のようなカテゴリに分類し、その監視を行う機能を実現することとした。

- アプリケーションのエラー
  - 処理の監視：処理が正しく実行されているかどうか、成立すべき条件等を要所要所で監視する。

- データの監視：データが破壊されていないか等を監視する。

■ リアルタイムOSのエラー

- 実行リソース割り当ての監視：タスクが期待された頻度で起動されているか等を監視する。
- メモリ割り当ての監視：メモリ割当エラーの発生を監視する。

■ ハードウェア(以下HW)のエラー

- 制御・監視対象HWの監視：制御・監視対象HWへの制御・アクセス時に成立すべき条件や、送受信されるデータ構造などを監視する。

これらのカテゴリ毎に安全性機能のアーキテクチャを決定した。以下に例を示す。

■ アプリケーションの処理の監視

アプリケーション処理を監視する機能である。典型的には重要な処理の前後に、前置条件や後置条件をチェックする機能をアプリケーションが呼び出し、その結果を規定されたエリアに書き込んでおく。検出機能はより高い優先度で周期起動され、その結果が異常であれば処理機能を呼び出す（図 2)。ここでチェック機能、検出機能、処理機能の具体的な内容はアプリケーション依存である。

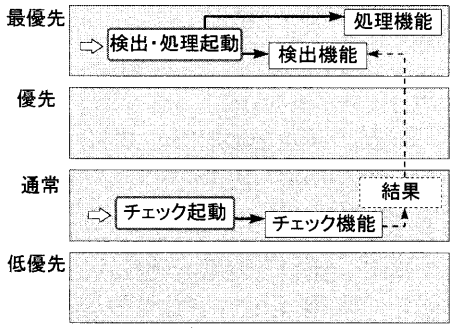

図2：アプリケーション処理の監視

■ アプリケーションのデータの監視

アプリケーションが保持するデータが破壊されていないかどうかを監視する。上記の処理の監視とは異なり、特定の処理コンテキストには依存しない。一種のヘルスチェックであるため、優先度の低い処理として監視を行う(図 3)。ただし一旦データの異常が発見されれば、最優先でエラー処理を起動する。ここでもチェック機能、処理機能の具体的内容はアプリケーション依存となる。

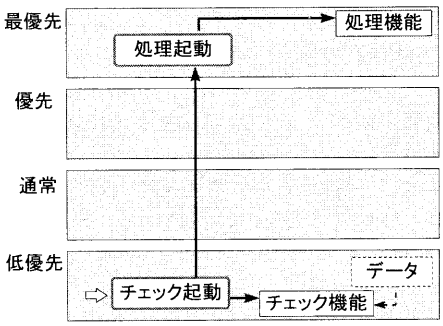

図3：アプリケーションデータの監視

■ 制御・監視対象HWの監視

制御対象となるHWに対して制御時点で成立していなければならない制約チェックや、制御命令、データのチェックなどを行う(図 4)。例えば同一の進路の上り下り両方向に対して信号機の進行現示が出されることは意味的にありえないので、そうした制約事項をチェックする。HWへ制御を出す部分は安全性の最後のチェックポイントとなるため、本監視機能の処理自体をさらに監視することにしたため、本監視機能を優先処理とし、本監視機能の監視機能を最優先処理とした。

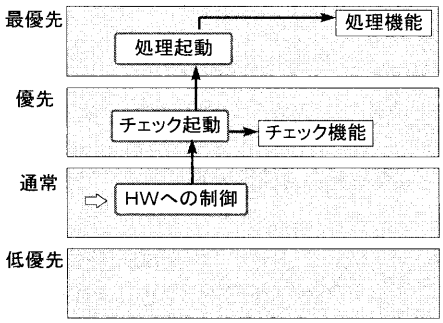

図4：対象HWの監視

以上のような個々のエラーに対応するアーキテクチャに対してそれぞれの実装方式を決定する。

3.2.3サービス機能アーキテクチャ

アプリケーションのサービス機能を実現するためのアーキテクチャである。ここでの構造は分析に用いる OOSE[4]でのモデル構造との対応付けをやりやすくするために(4章参照)、制御部、データ管理部、I/O 部の３つから構成される。今回試作したアプリケーションでは、サービス機能部分は、複数の制御部がデータフロー的に駆動するというアーキテクチャを採用した。

サービス機能はすべて基本制御アーキテクチャの通常処理部分に位置づけられる。したがって、通常処理部分に認められたメカニズムのみを利用してこうしたアーキテクチャを実現することになる。例えば今回の実装方式では、制御部はすべて同じ優先度を持ったタスクとして実現され、ラウンドロビンでスケジューリングされる。さらに割り込みなどの機能を利用することはできない。

また制御部、データ管理部、I/O 部は安全性機能アーキテクチャの制約を受ける。例えば制御部に対してはアプリケーション処理の監視機能が、I/O 部には制御・監視対象HWの監視機能が対応づけられる(図 5)。これらの安全性機能アーキテクチャの動作を攪乱しない形でサービス機能アーキテクチャを構築する必要がある。

	制御部	データ部	I/O 部
処理	○		
データ		○	
実行リソース	○		
メモリ割当	○		○
対象HW			○

図5：安全性機能との基本的な対応関係

3.3 プラットフォーム機能の抽出

プラットフォーム化の目的は、上述したアーキテクチャに従った安全性ソフトウェアの実現を容易にすること、また逆にそのアーキテクチャの枠組みを壊さずにアプリケーションを実現させることにある。

プラットフォーム機能の抽出に関する基本的な方針は、通常のアーキテクチャの場合と類似している。すなわち、そのプラットフォームが想定するアプリケーションのファミリーを明確にし、それらに基づいてホットスポットを見いだして何をプラットフォーム化すべきかを決定するというものである。

今回の想定ターゲットは、鉄道信号用ソフトウェアに代表される制御ソフトウェアであるが、例えば連動装置という具体的なアプリケーションのみを対象とするわけでない。従ってサービス機能アーキテクチャ自体はプラットフォームには含めないこととした。安全性機能アーキテクチャについては、前述したチェック機能、処理機能などはアプリケーション依存であり、安全性機能のホットスポットに相当する。したがってこれら以外の部分をプラットフォーム機能に含めることとした。基本制御アーキテクチャは、基本的には開発者が手を加えてはならない部分とし、すべてプラットフォーム中に組み込んだ。従って開発者がタスクの優先度を自分で設定したり、割り込み機能を用いることは原則的にできない。

4　アーキテクチャに基づく開発

上述したソフトウェアアーキテクチャを用いた開発を方向づけるために、開発の手順を整理した。本章では、その手順を概観する。

4.1 要求定義・分析

要求定義・分析の第一の目的は、アプリケーションのサービス機能を明確にすることにある。特に今回対象としたソフトウェアは、複雑なルールや制約に従った制御を行うため、厳密な分析が必須である。分析はほぼ OOSE に従い、要求定義ではユースケースの作成や問題ドメインオブジェクトの洗い出しを行い、分析では制御オブジェクト、エンティティオブジェクト、インタフェースオブジェクトを用いて分析モデルを作成する。

安全性に関しては、制御オブジェクトやエンティティオブジェクトの単位に、チェックすべき制約条件等を洗い出す。またエラー処理の内容を決めるために、安全側とはなにかを定義する。この作業はすべてのオブジェクトに対して網羅的に行なう必要はなく、絶対に起こってはならない状況など、明確に定義可能な部分のみに注目すればよい。

4.2 設計

設計段階では、基本的な実装方針を明らかにする。設計以降の作業は実装するターゲット環境や用いる言語などに依存する内容となる。特に今回対象としたソフトウェアは組み込みソフトウェアであり、オブジェクト指向での分析結果をどのように実装するかは、ターゲット環境に大きく左右される。今回はリアルタイムOS上でC言語を用いて実装したが、CPU 性能に余裕があったため、論理構造をそのまま保存する形で、制御オブジェクト（もしくは制御オブジェクト群）を制御部、エンティティオブジェクトをデータ管理部、インタフェースオブジェクトをI/O 部に対応づけた。

安全性に関しては、図 5に示したように制御部、データ管理部、I/O 部に対して提供される安全性機能が明確化されているので、分析段階で洗い出した安全性の要件を踏まえ、個別に行なうべきチェックやエラー処理の内容を明らかにする。ドメインエキスパートは、分析モデルとの対応づけの中で、どういう処理に対してどのようなエラーをチェックしようとしているのか等を確認することができる。この確認にはサービス機能の実現方法に関する知識は不要である。

さらにサービス機能アーキテクチャ(今回の例では前述したデータフローのアーキテクチャがそれに相当する)を決定する。ドメインエキスパートはこの内容を必ずしも理解する必要はない。

4.3 実装

サービス機能アーキテクチャ上の制御部、データ管理部、I/O 部の具体的な実装を行なう。今回の実装方法を図 6に示す。またメッセージのやりとりはメッセージキューや関数呼び出しを用いて実現した。

基本制御アーキテクチャ、安全性機能アーキテクチャからの制約があるため、アプリケーション部分に利用できるタスクの優先度は固定されており、それらはラウンドロビンでスケジューリングされるのみである。また割り込み等も一切利用できない。サービス機能アーキテクチャは、それらの枠組みの中で実現しなければならない。なおフラグの値としては 0 などの危険な値を利用してはならない等の局所的な技法に関しての考慮も必要となる。

	実装方式
制御部	タスク
データ管理部	データと関数群
I/O 部	データと関数群

図6：基本単位の実装方式

5 考察

本アーキテクチャは、安全性機能を組み込んだ基本単位(制御部等)に基づいてソフトウェア構造を構成するものと考えることができる。サービス機能はその基本単位をベースにして実現される。各基本単位がどのようなカテゴリのエラーに対してどのような安全性機能を備えているのかという知識さえあれば、実現の詳細に立ち入ることなく、安全性を確認することが可能となる。半面、サービス機能アーキテクチャは、安全性機能アーキテクチャの枠組みの中で実現することが強要されるため、実装上の制約がきつくなるというデメリットがある。本アーキテクチャの妥当性は、サービス機能と安全性機能のどちらを重視するかに依存して判断されるべきものである。

アーキテクチャを規定しようとしても、それを義務づけるなんらかのメカニズムがないと現実的には規定は困難である。今回のプラットフォームはアーキテクチャを実現するために必要となる機能を提供してはいるが、このアーキテクチャからの逸脱を強く制限する機能は持っていない。状況によっては、よりきつい規定メカニズムが必要になることもあろう。逆に、ガイドラインさえ示せば特別のプラットフォームなどを用意しなくても通常のリアルタイムOSさえあれば十分という状況も考えられる。こうした形態には絶対的な規定レベルが存在するわけではなく、個々の状況に応じた規定レベルが存在すると考えるべきであろう。

今回の開発手法ではオブジェクト指向で分析モデルまでを作成したが、組み込みシステムであるなどの理由から非オブジェクト指向で実装することにした。今回の試作は余

裕のあるCPUを用いたため、分析モデルの論理構造を崩さずに実装することができたが、実装制約が厳しくなれば、そうしたわかりやすい実装ストーリを提示できないかもしれない。安全性だけでなく、レスポンス、メモリサイズなど様々な非機能的要求を同時に考慮する必要がある場合には、アーキテクチャの決定はより困難になる。そうしたことへの対応は今後の課題である。

6 関連研究

近年ソフトウェアアーキテクチャやデザインパターンの研究が活発化しているが[1][2][10]、前述したようにその多くは修正容易性や拡張性などに関する考慮が主体であり、それ以外の非機能的な要求に対する考慮は十分ではない。そういう中で、本研究は安全性を重視したアーキテクチャの事例として位置づけられる。

デザインパターンはマイクロアーキテクチャに該当するものと考えられるが、よりグローバルなアーキテクチャのパターン化の検討も試みられている[1][10]。しかしながら現状は個々のパターンの蓄積、記述に重点がおかれており、パターンシステム[1]などの考え方はあるものの、複数のパターンの組み合わせ等に関する研究はこれからという段階である。我々はグローバルアーキテクチャ、マイクロアーキテクチャ間の依存関係に注目した開発手法等を検討しているが[5]、本稿での事例は、何をグローバルアーキテクチャと考えるかによって、ソフトウェアの構造や開発手法が大きな影響を受けることを示している。

なおソフトウェアの安全性実現のための代表的なアプローチとして、フォールトトレランスとフォールトアボイダンスの考え方がある。安全性機能アーキテクチャは、フォールトトレラントな機能を実現するためのアーキテクチャと考えられるが、ドメインエキスパートにとっての理解容易性や、開発者による設計品質のばらつきの減少など、フォールトアボイダンスの側面への効果までをねらったものである。なお安全性機能のプラットフォーム化という考え方は以前から提案されているが[3][6][7][8]、本研究はソフトウェアアーキテクチャおよび、それを想定した開発手法という観点からのアプローチである点に特徴がある。

7 おわりに

本稿では、安全性ソフトウェアのためのソフトウェアアーキテクチャとそれに基づいたプラットフォームや開発手法について報告した。今後より多様な非機能的要求に応じたソフトウェアアーキテクチャやその設計手法を検討していきたい。

参考文献

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

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

[3] Huang, Y., et.al: *Software Rejuvenation : Analysis, Module and Applications*, Proc. of FTCS-25, (1995).

[4] Jacobson, I., et.al: *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, (1992).

[5] 岸,他: 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).

[6] Leveson, N.G., et.al.: *Safety Assertions for Process Control Systems*, Proc. of FTCS-13, (1983).

[7] Leveson, N.G.: *Safeware*, Addison-Wesley, (1995).

[8] Russinovich, M., et.al,: *Fault-Tolerance for Off-The-Shelf Applications and Hardware*, Proc. of FTCS-25, (1995).

[9] Schmid, H.A.: *Creating the Architecture of a Manufacturing Framework by Design Patterns*, Proc. of OOPSLA'95, (1995).

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